

© 2020 Anjali Narayan-Chen

TOWARDS COLLABORATIVE DIALOGUE IN MINECRAFT

BY

ANJALI NARAYAN-CHEN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Associate Professor Julia Hockenmaier, Chair

Professor Heng Ji

Professor Dan Roth

Professor Martha Palmer, University of Colorado, Boulder

Abstract

This dissertation describes our work in building interactive agents that can communicate with humans to collaboratively solve tasks in grounded scenarios. To investigate the challenges of building such agents, we define a novel instantiation of a situated, Minecraft-based, Collaborative Building Task in which one player (**A**, the Architect) is shown a target structure, denoted *Target*, and needs to instruct the other player (**B**, the Builder) to build a copy of this structure, denoted *Built*, in a predefined build region. While both players can interact asynchronously via a chat interface, we define the roles to be asymmetric: **A** can observe **B** and *Target*, but is invisible and cannot place blocks; meanwhile, **B** can freely place and remove blocks, but has no explicit knowledge of the target structure. Each agent requires a different set of abilities in order to be successful at this task: specifically, **A**’s main challenges arise in the task of *generating* situated instructions by comparing *Built* and *Target*, while **B**’s responsibilities focus mainly on *comprehending* **A**’s situated instructions using both dialogue and world context. Both agents must be able to interact asynchronously in an evolving dialogue context and a dynamic world state within which they are embodied.

In this work, we specifically examine how well end-to-end neural models can learn to be instruction givers (i.e., Architects) from a limited amount of real human-human data. In order to examine how humans complete the Collaborative Building Task, as well as use human-human data as a gold standard for training and evaluating models, we present the Minecraft Dialogue Corpus, a collection of 509 conversations and game logs. We then introduce baseline models for the challenging subtask of Architect utterance generation, and evaluate them offline, using both automated metrics and human evaluation. We show that while conditioning our model on a simple representation of the world gives our model improved ability to generate correct instructions, there are still many obvious shortcomings, and it is difficult for these models to learn the large variety of abilities needed to be successful Architects in an entirely end-to-end manner. To combat this, we show that including meaningful, structured inputs about the world and discourse state as additional inputs – specifically, by adding oracle information about the Builder’s next actions, as well as enriching our linguistic representation with Architect dialogue acts – improves the performance of our utterance generation models. We also augment the data with shape information by pretraining 3D shape localization models on synthetically generated block configurations. Finally, we integrate Architect utterance generation models into actual Minecraft agents, and evaluate them in a fully interactive setting.

Acknowledgments

First and foremost, I would like to thank my advisor, Professor Julia Hockenmaier, for her tireless support and guidance throughout my Ph.D. When I first arrived at Illinois, fresh out of undergrad without the faintest clue about natural language processing, Julia reached out to me almost immediately and welcomed me into her group, no questions asked. Since then, while the field dramatically changed during my time as a student, I was lucky enough to have Julia to help carve my own path through it. I am both amazed and grateful that she not only encouraged this crazy idea of dialogue in Minecraft, but also helped us develop and frame it into a fully-fledged, fundamentally sound research problem. In addition to shaping me into a stronger, more detail-oriented researcher, she was also there to lend an ear when I needed personal or emotional support. My graduate school experience would not have been the same, and not nearly as enjoyable, without her.

I would also like to thank the rest of my committee: Professor Heng Ji, Professor Dan Roth, and Professor Martha Palmer, for their valuable time and comments, and for helping me take a step back and view the work in this thesis from a much broader lens. I am grateful for their feedback and insights during the prelim and final exams which have helped shape this work.

I would like to specially thank my undergraduate advisor at the University of Wisconsin-Madison, Professor Jude Shavlik, for opening my eyes to research and igniting my passion for artificial intelligence. Little did I know what a journey I was signing up for when I walked into his class to learn about perceptrons! I am forever grateful that Jude took me under his wing as a budding researcher in the summer of 2013. I will always fondly remember that life-changing summer Liqi Xu and I spent in Jude’s lab in the UW-Madison Biostatistics building, coding up naive Bayes models to play Angry Birds, and it is somehow fitting that this thesis continues my research trend of AI in video games, albeit in a very different way.

I gratefully acknowledge the support of the Defense Advanced Research Projects Agency (DARPA) Communicating with Computers Program Contract W911NF-15-1-0461 for both funding this work and providing me the opportunity to work with such a talented group of collaborators.

Specifically, this work would not have been at all possible without my lab partner-in-crime, Prashant Jayannavar, “the best labmate ever” (in his own words). Without the countless hours we spent together designing Minecraft clients, running data collection, throwing around ideas, implementing models, writing papers until 5AM, and, of course, hacking to-

gether last-minute demos, this work would not exist. I am eternally grateful.

I would also like to thank the various other members of Julia's lab that I have had the opportunity to overlap and interact with: Yonatan Bisk, Micah Hodosh, Alice Lai, Chris Cervantes, Ryan Musa, Rajarshi Haldar, and Marc Canby. I am grateful to have been able to learn from my peers and share in their experience. In particular, I would like to thank Yonatan and Alice for their invaluable advice and guidance in my later years.

I am also lucky to have the support of friends I made while at Illinois: Colin Graber, Lauren Kolkman, Chris Sartain, Ian Dace, Michael Atten, Garrett Wirka, and Xin Ding, among others. I will always cherish the times we spent together, attending and sometimes winning trivia, solving escape rooms, and convening for Dungeons and Dragons sessions (although, as of this writing, it is still unclear whether or not our campaign will take longer than my Ph.D. did) as an essential part of the graduate school experience. I would like to specially thank Alex Morales for putting up with me, and supporting me, during the most difficult parts of my Ph.D. journey, despite also going through the same stresses of his own.

Finally, I would like to thank my family for their unwavering support during my Ph.D. I am especially indebted to my parents for keeping their stubborn faith in me throughout my ups and downs in school, reminding me that *I can do this*, and doing so despite my insistence to stay far away from supply chain optimization and instead explore the (to them, taboo) subjects of machine learning and video games. I am grateful that my parents always reminded me of the light at the end of the tunnel, especially at times when I could not see it myself. Thank you for everything.

Table of Contents

Chapter 1	Introduction	1
1.1	Motivation	1
1.2	Thesis Statement	2
1.3	Background	4
1.4	The Minecraft Collaborative Building Task	6
1.5	Overview of this Dissertation	13
Chapter 2	The Minecraft Dialogue Corpus	15
2.1	Implementation Details	15
2.2	Collecting the Minecraft Dialogue Corpus	18
2.3	Data Statistics and Analysis	20
2.4	Other Uses of the Corpus	25
2.5	Conclusion	26
Chapter 3	Architect Utterance Generation	27
3.1	Background	28
3.2	Task Definition	32
3.3	Seq2Seq Architect Utterance Model	34
3.4	Simple World State Representations: Block Counters	34
3.5	Experimental Setup	38
3.6	Results and Analysis	39
3.7	Block Counters v2	49
3.8	Using Oracle Next Actions	54
3.9	Conclusion	57
Chapter 4	Adding Linguistic Structure: Dialogue Acts	59
4.1	Background	60
4.2	Adding Dialogue Acts to the Minecraft Dialogue Corpus	63
4.3	Dialogue Act Classification	67
4.4	Next Dialogue Act Prediction	71
4.5	Dialogue Acts for Architect Utterance Generation	79
4.6	Combining Dialogue Acts with Oracle Next Actions	88
4.7	Conclusion	93
Chapter 5	Representing the World: Shape Localization	95
5.1	Synthetic 3D Block Configurations	96
5.2	Shape Localization	99
5.3	Convolutional Models for Shape Localization	102
5.4	Results and Analysis	105
5.5	Localizing Shapes in Minecraft Target Structures	112

5.6	Towards a CNN-Based Architect	116
5.7	Conclusion	120
Chapter 6	Designing an Interactive Architect	121
6.1	Designing an Interactive Architect	121
6.2	Interactive Architect Examples	126
6.3	Conclusion	136
Chapter 7	Conclusion	137
7.1	Comparison to Concurrent Work	138
7.2	Future Work	142
Appendix A	Example Dialogues from the Corpus	145
Appendix B	Term-Specific Word Lists	150
Appendix C	Synonym Map	151
Appendix D	Hyperparameter Configurations	155
D.1	Block Counters	155
D.2	Block Counters v2	156
D.3	Using Oracle Next Actions	157
D.4	Dialogue Acts for Architect Utterance Generation	157
D.5	Combining Dialogue Acts with Oracle Next Actions	158
References	160

Chapter 1: Introduction

1.1 MOTIVATION

There has been longstanding interest in developing interactive agents that can communicate with humans to collaboratively solve tasks in grounded scenarios (e.g. SHRDLU [1]). In these scenarios, in order for an agent to successfully communicate in natural language and collaborate with humans in the real world, they must not only be able to recognize and describe the immediate world around them, but they must also be able to ground the concurrent discourse to that world to effect changes within it. However, such situated dialogue poses challenges that go beyond those currently studied in standard dialogue tasks within natural language processing (NLP) literature. For example, slot-value filling tasks performed by standard dialogue systems can be represented using flat meaning representations that can straightforwardly be grounded to APIs [2, 3, 4], while chatbots typically operate in unconstrained dialogue scenarios without clearly-defined goals [5, 6, 7]; other dialogue scenarios may involve further grounding to images or videos, such as in so-called visual dialogue where users talk about a static image [8] or video-context dialogue where users interact in a chat room while viewing a live-streamed video [9], but do not deal with the challenges that arise from embodied agents interacting in and manipulating a dynamic environment.

In contrast to these unconstrained and ungrounded scenarios, instruction giving/following tasks are common testbeds that showcase situated language. In these tasks, instruction givers need to be able to refer to real-world objects in ways that depend on the current position of the speakers as well as changes in the environment. Instruction followers, on the other hand, should not only be able to engage in rich natural language dialogue with their human conversation partners, but also to ground that dialogue to physical objects, and execute instructions in the real world. In practice, the ability to understand and interact using situated language is especially important for embodied robots that are intended to be used as teammates for human partners. However, relevant efforts in robotics have largely focused on single-shot instruction following, and are mostly constrained to simple language [10, 11] with limited resources [12, 13, 14]. Due to the expense of actual human-robot communication, simulated environments that allow easier experimentation are commonly used [15, 16, 17].

Therefore, a necessary first step towards building interactive agents is to design a situated, goal-oriented dialogue scenario that provides the benefits of easier experimentation afforded by simulated environments, but still realistically captures the challenges of a grounded dialogue task. To this end, we define the situated **Minecraft Collaborative Building Task**,

that involves two players: an Architect (**A**) instructs a Builder (**B**) to construct a target structure out of multicolored building blocks in the 3D virtual environment provided by the Minecraft gaming platform. The Collaborative Building Task situates a dialogue task, intentionally designed to be asynchronous and asymmetric with few constraints on the language, in a simulated, 3D Minecraft environment within which speakers with constantly changing perspectives must refer to and manipulate a constantly changing world.

1.2 THESIS STATEMENT

In this thesis, we explore how to build automated agents that can engage in asynchronous, situated natural language dialogues with humans in order to collaboratively build structures in 3D virtual environments. We show how such environments make it possible to study situated language generation and understanding, especially in the context of construction tasks, in which agents must be able to refer to and effect changes in a dynamic environment. Specifically, we design an instantiation of a situated task by simulating a collaborative building task in Minecraft. This allows us to make some simplifying assumptions about the ease with which agents can perceive or interact with the world which don't hold in real physical environments. Furthermore, motivated by the recent, paradigm-shifting successes of end-to-end neural models for many NLP tasks, and by the fact that such models can be trained on raw data without rich linguistic annotations, we examine how well end-to-end neural models can learn to be instruction givers from a limited amount of real human-human data, and find that while these models can learn to process the world to a small extent, it is difficult for them to learn the large variety of abilities needed to be successful conversation partners in this domain in an entirely end-to-end manner. We show that these models can instead benefit from including additional meaningful, structured inputs about the discourse and world state.

Specifically, in this dissertation, we build interactive agents that can communicate with humans to collaboratively solve tasks in grounded scenarios by first defining a novel instantiation of a situated, Minecraft-based, Collaborative Building Task in which one player (**A**, the Architect) is shown a target structure, denoted *Target*, and needs to instruct the other player (**B**, the Builder) to build a copy of this structure, denoted *Built*, in a predefined build region. While both players can interact asynchronously via a chat interface, we define the roles to be asymmetric: **A** can observe **B** and *Target*, but is invisible and cannot place blocks; meanwhile, **B** can freely place and remove blocks, but has no explicit knowledge of the target structure. Each agent therefore requires a different set of abilities in order to be successful at this task. Specifically, **A**'s main challenges arise in the task of *gener-*

ating situated instructions. **A** must align *Target* and *Built* in order to compare them and determine next steps, since *Built* can be a rotated and translated copy of *Target* within the build region. Beyond providing clear instructions, which should appropriately make use of **B**’s perspective and block actions, **A** must also be able to describe *Target* in terms of substructures such as *rows*, *columns*, *diagonals*, *staircases*, etc. as well as higher-level terms such as *giraffe* or *table*. Additionally, **A** should be able to identify when mistakes are made and generate appropriate corrections, as well as answer **B**’s questions appropriately when **B** requests clarification or verification. Meanwhile, **B**’s responsibilities focus mainly on *comprehending* situated instructions using both dialogue and world context; these challenges are amplified as **A**’s instructions are spread across turns, **B**’s perspective changes with movement, and **B**’s actions change the environment. **B** must also grapple with the challenge of learning to predict sequences of block actions within a large action space, in which floating blocks require nonmonotonic action sequences where a placement is followed by a removal, and training sequences are noisy since human **B** players are prone to mistakes and misunderstanding. Furthermore, to close the loop, **B** needs to know when and how to continue and clarify the dialogue, by asking appropriate verification or clarification questions, giving suggestions, and extrapolating from existing context.

In this thesis, we specifically examine how well end-to-end neural models can learn to be instruction givers (i.e., Architects) in this domain. To this end, in order to examine how humans complete the Minecraft Collaborative Building Task, as well as use human-human data as a gold standard for training and evaluating models for this task, we present the Minecraft Dialogue Corpus, a collection of 509 conversations and game logs of humans completing this task. These human-human interactions highlight the various challenges associated with being an Architect: namely, that they not only must be able to provide clear instructions grounded to the Builder’s perspective, but also compare built and target structures to determine next steps, describe the target structure in meaningful terms, and respond to the Builder’s actions and queries appropriately in a context-sensitive manner. To examine how well neural models can address these challenges, we therefore introduce baseline models for the challenging subtask of Architect utterance generation and evaluate them offline using both automated metrics and human evaluation. We show that while conditioning our model on a simple representation of the world gives our model improved ability to generate correct instructions, there are still many obvious shortcomings, and it is difficult for these models to learn the large variety of abilities needed to be successful Architects in an entirely end-to-end manner. To combat this, we show that including meaningful, structured inputs about the world and discourse state as additional inputs – specifically, by adding oracle information about the Builder’s next actions, as well as enriching our linguistic representation with Archi-

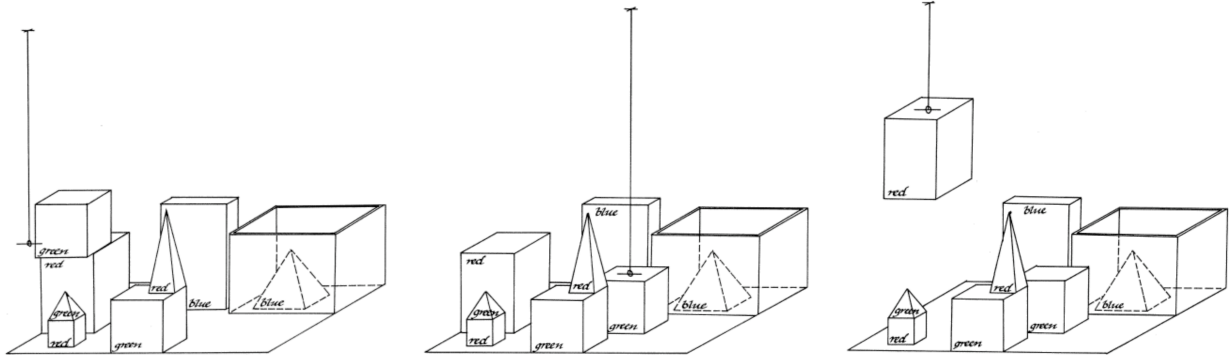


Figure 1.1: The original Blocks World of Terry Winograd’s SHRDLU [1]. This sequence depicts the series of actions taken by SHRDLU to execute the instruction “*pick up a big red block,*” in which the robot first moves a smaller green block out of the way before picking up the larger red one.

tect dialogue acts – improves the performance of our utterance generation models. We also augment the data with shape information by pretraining 3D shape localization models on synthetically generated block configurations and analyze their predictions on real Minecraft target structures. Finally, we integrate Architect utterance generation models into actual Minecraft agents and evaluate them in a fully interactive setting. In doing this, we show that while we have made significant progress towards our ultimate goal of building interactive agents, there is still much work that remains to be done, and we propose several promising avenues for future work.

1.3 BACKGROUND

Blocks World is a well-established domain for natural language interaction and planning in which agents construct structures using physical blocks that both appear and behave realistically. The Blocks World was originally conceived in 1968 as part of Terry Winograd’s SHRDLU [1] (Figure 1.1), an early situated language understanding and generation system that enabled users to converse in normal English with a computer that could answer questions about, execute commands in, and accept information regarding the state of a simplified virtual Blocks World that obeyed basic physics. In this scenario, SHRDLU, equipped with a robot arm and eye, must solve planning problems to complete tasks involving physical constraints (such as retrieving blocks out from underneath others), as well as talk about the state of the world and its understanding of it using natural language. To illustrate this, we include a snippet of an example dialogue between a human user, **H**, and SHRDLU, **S**, from Winograd’s thesis below:

H: *pick up a big red block.*
 S: *OK.*
 H: *grasp the pyramid.*
 S: *I don't understand which pyramid you mean.*
 H: *find a block which is taller than the one you are holding and put it into the box.*
 S: *By "it", I assume you mean the block which is taller than the one I am holding.*
 S: *OK.*
 H: *what does the box contain?*
 S: *The blue pyramid and the blue block.*
 H: *what is the pyramid supported by?*
 S: *The box.*
 H: *how many blocks are not in the box?*
 S: *Four of them.*

In designing SHRDLU to operate robustly in the Blocks World, Winograd's thesis explored the hypothesis that a computer must *understand* a subject in order to converse about it reasonably in natural language. While the Blocks World may seem deceptively simple, Winograd demonstrated that, for an agent to be successful, it must grapple with the problems of understanding and generating natural language, representing and retaining knowledge in memory, planning and executing solutions, and grounding concepts to discourse and world context, all within a unified framework. However, Winograd accomplished this by designing SHRDLU's syntactic parsing, heuristic semantic understanding, and planning components using an extensive set of domain-specific handwritten rules and knowledge, resulting in a system that, though impressive when applied to constrained scenarios such as the one above, was also very brittle, and could only handle a small set of hard-coded concepts. While the Blocks World is a toy domain, the challenges that come with building interactive systems that can instead *learn* to be successful in this domain by incorporating the same variety of abilities are both realistic and complex. We therefore define a novel instantiation of a situated dialogue task in our rendition of Blocks World that allows us to investigate these aspects.

Minecraft (<https://minecraft.net/>) is a popular sandbox video game developed by Mojang Studios. In the full game, players control avatars to navigate in a blocky, procedurally-generated 3D world and can build structures, craft tools and items, and harvest raw materials, such as blocks of dirt, stone, and wood, from natural resources such as trees or mountains (see Figure 1.2 for an example). Minecraft worlds support both single-player



Figure 1.2: A Minecraft world. [18]

and multiplayer gameplay, as well as a variety of different game modes, including survival, creative, and spectator modes. Players can freely move, jump and fly, and they can choose between first- or third-person perspectives. Camera angles can be smoothly rotated by moving around or turning one’s avatar’s head up, down, and side-to-side, resulting in a wide range of possible viewpoints. Additionally, the Java edition of Minecraft is particularly well known for its flexible API that can be modified by independent developers, allowing for unique extra features, or *mods*, to be added to the base game.

Minecraft is a platform of growing interest for AI researchers. In particular, our work extends Microsoft’s Project Malmo [19], an AI research platform implemented as a Minecraft mod that provides an abstraction layer on top of Minecraft. The platform was originally designed to further research in wide variety of areas, including robotics, computer vision, reinforcement learning, planning, multi-agent systems, and others. Malmo enables human users and automated agents to interact in customized Minecraft worlds and provides an API for saving and loading game states. Combining Malmo’s extended functionality with Minecraft’s inherently block-based worlds provides an ideal setting to simulate the Blocks World.

1.4 THE MINECRAFT COLLABORATIVE BUILDING TASK

As a first step towards building interactive agents that can communicate and collaborate in real-world scenarios, we design a situated, goal-oriented dialogue scenario that realistically captures the challenges of a grounded dialogue task in a dynamic environment. Specifically, we define an instantiation of a situated dialogue task in which we can investigate the following questions:

- Given the changing state of the world, how do you identify and describe what to do

next in an intuitive way?

- Given the changing positions of the speakers, how should you identify the locations of objects in the world?
- How do dialogue and action history influence how you communicate the next steps?
- How do these interactions occur in a natural, real-time conversation setting?

To this end, we define the Minecraft Collaborative Building Task, in which two players are tasked with constructing a target structure out of multicolored building blocks. In this section, we first describe how we have re-imagined a Blocks World scenario in Minecraft, then formally define the Collaborative Building Task and highlight its associated challenges.

1.4.1 Blocks World in Minecraft

In our Blocks World scenario, structures are built out of 6 types of equally-sized colored blocks within a fixed, nondescript $11 \times 9 \times 11$ build region. We designed our blocks in this way specifically to disallow users from referring to specific blocks by uniquely identifiable names, as well as to allow us to design target structures with unique styles of substructures demarcated by color (e.g. blue rows, red towers, orange *T*-shapes, etc.). Furthermore, by design, the build region and its surroundings contain no identifiable landmarks, allowing for multiple valid perspectives without enforcing absolute cardinal directions (e.g. “north”).

However, our scenario differs from traditional Blocks World in a few key ways. First, Minecraft blocks can only be placed on a discrete 3D grid. As a result, blocks in Minecraft cannot be positioned such that they overlap multiple other blocks or be rotated at non-right angles. In this aspect, our scenario is simpler than traditional Blocks World. Second, Minecraft blocks do not need to obey gravity. That is, they can be put anywhere as long as one of their sides touches another block or the ground, allowing for a block to “hang” off the sides of adjacent neighbors; these neighbors can later be removed, allowing the second block (and any structure supported by it) to “float.” Furthermore, our task also features asymmetric roles and levels of information between the two speakers, but focuses on the creation of structures rather than navigation around and manipulation of existing ones; this allows us to investigate how communication evolves as the environment is dynamically changed by the speakers, as well as how to refer to objects that already exist and those not yet built.

Finally, a key difference between our Minecraft recreation of Blocks World to traditional Blocks World is in how the agents are situated in the world. Winograd’s SHRDLU involves a

robot arm that has global range of movement over a build region, which is viewed from a fixed perspective. Minecraft instead manifests players as moving avatars that navigate within the world using continuous movement and who are themselves no taller than a couple of blocks. Instead of as a universal robot arm, our scenario realizes the instruction follower as an avatar with a first-person perspective who walks around the structure in the build region and flies above the ground in order to reach taller blocks; the instruction giver, implemented as an invisible Minecraft “spectator,” has similar freedom of movement and can watch the follower from any number of angles, including a third-person view snapped to a location behind the follower. This enables instruction givers to use spatial relations grounded specifically to the follower’s perspective, such as “*extending in the direction you’re currently facing, and upwards*” to describe where a block should be placed. The added flexibility of perspectives in our scenario thus leads to a much more dynamic context than what is traditionally offered in the Blocks World domain.

1.4.2 Task Definition

Having first described how we reimaged the Blocks World in Minecraft, we now define our novel situated goal-oriented dialogue task designed for players in the Minecraft Blocks World.

We define the Collaborative Building Task as a two-player game between an Architect (**A**) and a Builder (**B**). **A** is given a target structure (*Target*) and has to instruct **B** via a text chat interface to build a copy of *Target* on a given build region. **A** and **B** can communicate back and forth via chat throughout the game (e.g. to resolve confusions or to correct **B**’s mistakes). **B** is given access to an inventory of 120 blocks of six given colors that it can place and remove. **A** can observe **B** and move around in its world, allowing it to provide instructions from varying perspectives. But **A** cannot move blocks, and remains invisible to **B**. The task is complete when the structure built by **B** (*Built*) matches *Target*, invariant to translations within the horizontal plane and rotations about the vertical axis. *Built* also needs to lie completely within the boundaries of the predefined build region.

Example We refer to Figure 1.3 for an example of game progression in the Collaborative Building Task. The dialogue snippet is taken from a point towards the beginning of the game in which **A** instructs **B** to build the orange 7 and yellow 6 structures that appear in *Target*. After some initial greetings and a description of the target – “*this is a bunch of giant block numbers ... start with orange and stop waving at me*” – **A** begins instructing **B** to build the stem of the 7:

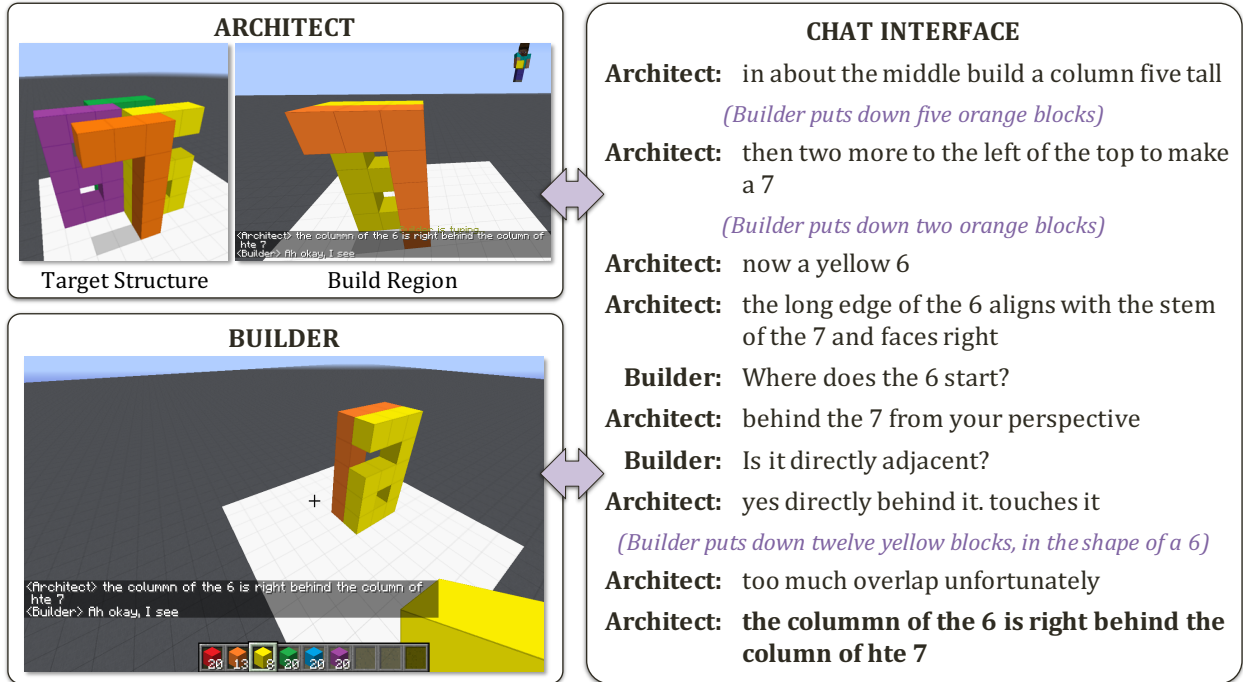


Figure 1.3: In the Minecraft Collaborative Building Task, the Architect (**A**) has to instruct a Builder (**B**) to build a target structure. **A** can observe **B**, but remains invisible to **B**. Both players communicate via a chat interface. (NB: We show **B**'s actions in the dialogue as a visual aid to the reader.)

A: in about the middle build a column five tall

Already, the instruction taken in isolation is underspecified, as the color of the column, *orange*, is only mentioned in the previous **A** utterance. Additionally, *in about the middle* is slightly ambiguous, and is left here to **B**'s interpretation. This is not necessarily a problem: the instruction is sufficient as long as the final built structure lies within the bounds of the build region. Despite this uncertainty, **B** executes this successfully, so **A** continues by providing another instruction that is highly dependent on **B**'s last actions and perspective:

A: then two more to the left of the top to make a 7

A not only references the previous column implicitly, but also uses **B**'s perspective as the frame of reference for the spatial relation *left of the top [of the column]*, all while referring back to a higher-level description of the target, a 7. **B**, possibly using common knowledge of what a 7 looks like as an aid, grounds and executes this next instruction successfully, so **A** continues by describing how the next shape, a 6, should align to the newly-constructed 7 by relating properties of the shapes rather than using block-wise or column-wise instructions:

A: *now a yellow 6*

A: *the long edge of the 6 aligns with the stem of the 7 and faces right*

But here, **A**'s instructions are less clear to **B**. Instead of building blocks, **B** asks a series of clarifying questions in order to relate the 6's location to the 7, to which **A** responds with further information:

B: *Where does the 6 start?*

A: *behind the 7 from your perspective*

B: *Is it directly adjacent?*

A: *yes directly behind it. touches it*

This exchange is rife with pronominal references to both the existing 7 and the yet-to-be-built 6, after which **B** deems the ambiguity resolved and proceeds to build the entire 6 in one go, taking cues from the dimensions of the 7 to guide their actions. But unfortunately, **B** has misinterpreted **A**'s instructions, so **A** must now intervene with an appropriate correction:

A: *too much overlap unfortunately*

A: *the columnn of the 6 is right behind the column of hte 7*

The above example shows some of the specific challenges of this task. Although human players were able to complete each structure successfully, this task is not trivial. At every turn, **A** must compare a constantly-evolving *Built* structure against the *Target* in order to correctly describe next steps. When doing so, **A** often provides instructions that they think are sufficient, but leave **B** still clearly confused, indicated either by **B**'s lack of initiative to start building or a confused response. Once a multi-step instruction is understood, **B** also needs to plan a sequence of steps to follow that instruction; in many cases, **B** chooses clearly suboptimal solutions, resulting in large amounts of redundancy in block movements. A misinterpreted **A** instruction may also lead to a whole sequence of blocks being misplaced by **B** (either due to miscommunication, or because **B** made an educated guess on how to proceed) until **A** decides to intervene (in the example, this can be seen with the built yellow 6). **A** could also misinterpret the target structure, giving **B** incorrect instructions that would later need to be rectified. This illustrates the challenges involved in designing an interactive agent for this task: the Architect needs to provide clear instructions, the Builder needs to identify when more information is required, and both agents may need to design efficient plans to construct complex structures.

More importantly, the above example also showcases the broader challenges of situated dialogue that we wish to investigate as outlined at the beginning of this section, and highlights some of the specific aspects of each of **A**'s and **B**'s asymmetric roles. As the world

state evolves throughout the dialogue, **A**’s instructions towards the goal similarly evolve, referring directly to previously-built substructures and adapting to situations in which errors have been made. Due to **B**’s constantly-changing perspective, **A** almost always involves **B**’s frame of reference when referring to locations in the world, and **B** must be able to adaptively comprehend **A**’s instructions as their perspective and world changes. Retaining a memory of past utterances and actions is essential to grounding future interactions, which might otherwise appear ambiguous. And, lastly, the intentional lack of strict turn-taking structure allows the interaction to occur in real-time, allowing for the realistic possibility of interruptions and crossed wires. The Minecraft Collaborative Building Task therefore enables us to study these important aspects of situated dialogue in human-human interactions.

1.4.3 Differences from Prior Work

Dialogue tasks Our work differs from several flavors of existing human-human corpora for various dialogue tasks. Specifically, our work investigates the challenges of grounding language to the real-world in task-oriented scenarios. In contrast, a large area of focus in prior work includes ungrounded, unconstrained, chit-chat dialogues. A large number of spontaneous dialogues can be automatically scraped from social media platforms, such as posts and replies on Twitter [5], or from forums, such as comment threads on Reddit [6]. Chit-chat can also be constrained to a particular domain, such as Ubuntu tech support scraped from IRC channel logs [7], or interactions pulled from fictional sources such as movie dialogues [20, 21]. While generally ungrounded, some chit-chat corpora feature conversations grounded to various knowledge sources, such as Wikipedia or news articles [22, 23] or documents [24]. However, without a clear definition of task completion, automatically evaluating response generation models in these unsupervised scenarios is not straightforward [25].

Alternatively, while there has also been a focus on various goal-oriented dialogue tasks, such as slot-filling tasks that involve a dialogue agent who is tasked with helping a user achieve goals like acquiring information or booking accommodations, the challenge of grounding language in these scenarios is much simpler than in our work. In these settings, a pre-defined set of domain-specific slots are filled over the course of a conversation and used to fulfill a user’s needs. Examples of these include transcribed Skype calls of tourist information as used in Dialog State Tracking Challenges [2, 3] or the large-scale MultiWOZ corpus [4] collected via Amazon Mechanical Turk in which humans play the role of the machine in a Wizard-of-Oz setup. Some corpora feature goal-oriented dialogues grounded to specific APIs that are defined by schemas [26] or used by underlying knowledge bases [27]. In contrast, the structured knowledge and context-aware world grounding required by the Minecraft Collab-

orative Building Task, whose instructions cannot be sufficiently represented using a simple combination of slots and values and grounded to flat API calls, are much more complex.

Finally, there are also dialogue tasks that require reasoning over multimodal inputs, but these tasks generally address dialogue interactions in static environments, while our work addresses such interactions in dynamic ones that can be influenced by the speakers themselves. Visual Dialog [8] involves users talking about a static image, while video-context dialogue [9] features a large amount of users interacting within a chat room while viewing a live-streamed video. The MMD benchmark dataset [28] features a large number of multimodal conversations, involving both text chat and images, between shoppers and salespeople in a retail domain. However, these multimodal scenarios do not embody the speakers in their environments, and these environments are not influenced by the actions of the speakers. While users the Collaborative Building Task must also talk about a grounded, multimodal medium (in our case, the 3D Minecraft world), our users are additionally required to dynamically change that world by manipulating objects within it, resulting in a much more variable context.

Situated language There have been a variety of prior situated language testbeds, commonly framed as instruction following for navigation tasks, which have inspired our work. The HCRC Map Task [29] tasks two speakers, sitting across from each other and each given slightly differing schematic maps, with successfully navigating a route through the map using verbal communication. Only one speaker sees a planned route on their map, and has to guide the other speaker to follow that route on a slightly different map. Another scenario includes the maze navigation task of MacMahon et al. (2006) [30], who design a corpus of route instructions from human instructors to human followers in large-scale indoor environments. The SPACEBOOK corpus [31] features dialogues collected in a Wizard-of-Oz setup, in which a human, pretending to be a tourist physically walking around the city of Edinburgh, converses with a human Wizard posing as a tour guide. Similar to our scenario, the GIVE challenge [15] tasks natural language generation systems with generating natural language instructions to guide a human user to perform tasks and find treasure in a virtual 3D environment. We build on this prior work by designing a similar situated environment to investigate how to build such interactive agents, but our work differs from these previous scenarios mainly in that we focus on construction rather than navigation tasks, specifically to investigate the implications and added complexity of having new and evolving entities in the world, and that we do not explicitly map utterances to symbolic representations, as most prior testbeds do.

Prior approaches to instruction comprehension typically take a semantic parsing approach. Chen and Mooney (2011) [32] build a semantic parser that learns to parse natural lan-

guage instructions to navigation plans involving action sequences. Artzi and Zettlemoyer (2013) [33] also learn a semantic parser for route instructions, but use a weakly-supervised approach involving a weighted Combinatory Categorical Grammar lexicon. Semantic parsing components enable human-robot understanding by mapping utterances to grounded Spatial Description Clauses [11, 34], or to expressions in a logic-based Robot Control Language [35], or to knowledge bases of facts about the environment [12]. Some approaches to interactive robot design combine these architectures with physical robot exploration to enable online learning [36, 37]. As we do not define symbolic representations for our utterances, we cannot directly use a semantic parsing approach; instead, we map **A** utterances directly to **B** action sequences that can be executed directly in the world. We explore this task in detail in Jayannavar et al. (2020) [38], but not in this thesis.

1.5 OVERVIEW OF THIS DISSERTATION

This dissertation is organized as follows:

Chapter 2: We present the Minecraft Dialogue Corpus, a collection of 509 conversations and game logs of humans completing this task, and provide detailed qualitative analysis of the resulting game logs. The work in this chapter is joint work with Prashant Jayannavar in our ACL 2019 paper [39].

Chapter 3: As a first step towards building fully interactive Architect agents, we define the challenging subtask of Architect utterance generation. We design and evaluate a series of end-to-end neural models on this task, and show that we achieve improved results by adding simple world representations to these models. The work in this chapter up until Section 3.7 is joint work with Prashant Jayannavar in our ACL 2019 paper [39].

Chapter 4: In order to add richer linguistic structure to our data and models, we annotate Architect utterances with their dialogue acts. We show preliminary results for modeling these dialogue acts in isolation, and also show that using them in downstream Architect utterance models yields improved results on the utterance generation task.

Chapter 5: We explore how to build better representations of the world state by pretraining convolutional neural models on synthetically generated data to identify the locations of elementary shapes in the Minecraft grid. We then show preliminary results of applying these pretrained models to real Minecraft target structures.

Chapter 6: We integrate our Architect utterance models into actual Minecraft agents and present them in a fully interactive setting. We highlight strengths and weaknesses of our model’s performance in real interactive settings.

Chapter 2: The Minecraft Dialogue Corpus

In order to examine how humans complete the Minecraft Collaborative Building Task, as well as use human-human data as a gold standard for training and evaluating models for this task, we first collect the **Minecraft Dialogue Corpus**, a collection of game logs of humans completing the Collaborative Building Task. This chapter presents a deep dive into further implementation details of recreating the Blocks World in Minecraft in Section 2.1 and how we collected the Minecraft Dialogue Corpus in Section 2.2. Finally, we walk through in-depth visual examples and analysis of the resulting game logs in order to further showcase the challenges of the Collaborative Building Task in Section 2.3.

2.1 IMPLEMENTATION DETAILS

In our Blocks World scenario, structures are built entirely out of 6 types of equally-sized colored blocks (red, orange, yellow, green, blue, and purple) whose faces are identical and featureless (and thus have no predefined orientation). Furthermore, we defined a fixed, nondescript $11 \times 9 \times 11$ build region, outlined using a white square floor along the xz plane, within which structures are built in order to efficiently constrain the size of the recorded game worlds. We also added some minor aesthetic enhancements, such as removing Minecraft-specific animations for destroying and harvesting blocks, to further distinguish the Blocks World scenario from traditional Minecraft gameplay.

Blocks in Minecraft do not obey gravity. That is, they do not need to be placed on the ground or on top of another block, but can be put anywhere as long as one of their sides touches another block or the ground, allowing for a block to “hang” off the sides of adjacent neighbors. Furthermore, the neighboring block can later be removed, allowing the second block (and any structure supported by it) to “float.” Figure 2.1 showcases the different

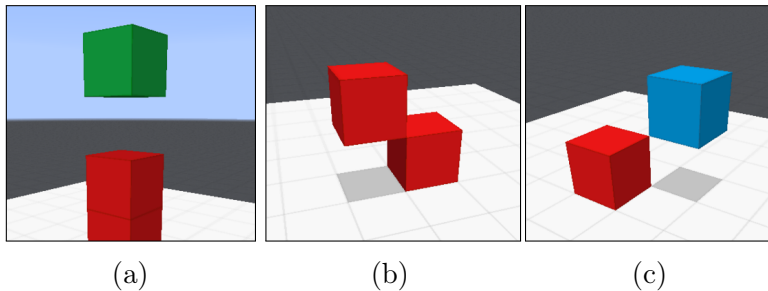


Figure 2.1: Types of floating blocks in Minecraft. Suspended blocks (a) and planar diagonals (b) require 1 placeholder block, while 3D diagonals (c) require 2 placeholder blocks.

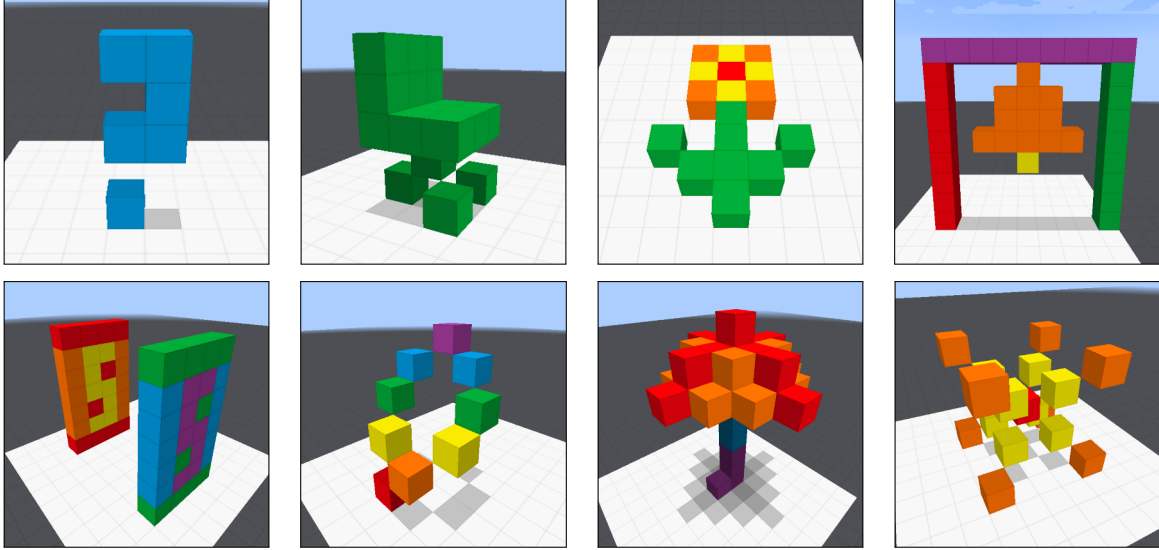


Figure 2.2: Sample target structures in the Minecraft Dialogue Corpus. From left to right, they can be described as follows: 1st row: blue question mark, green chair, colorful flower, and belltower; 2nd row: mirrored embedded 5s, rainbow lasso, umbrella, and 3D X.

kinds of floating blocks possible in Minecraft. These result in more complex, non-monotonic action sequences than those of traditional Blocks World, as players need to identify when such supporting (or “placeholder”) blocks need to be added or removed, as well as allow for complex, realistic-looking structures to be built without requiring a large number of blocks to satisfy physical constraints.

Defining target structures We manually designed 150 target structures in Minecraft of varying complexity (min. 6 blocks, max. 68 blocks, avg. 23.5 blocks), with an additional 4 simple structures for warm-up purposes. To showcase differing aspects of the target structures across a sample subset, we refer to Figure 2.2. Structures could be monochromatic (e.g. blue question mark or green chair), but were more often multicolored (avg. 3.42 colors per structure). Target structures ranged from planar drawings flat on the ground (e.g. colorful flower) to planar drawings in the y plane (e.g. belltower or mirrored embedded 5s) to complicated 3D constructions (e.g. umbrella), sometimes with a large number of floating blocks (e.g. rainbow lasso, which consists almost entirely of floating blocks). Some target structures were designed to mimic real-world objects, such as furniture or animals, while others consisted of abstract designs (e.g. 3D X). Finally, some structures exhibited repeating substructures that could be referred back to as later substructures were built (e.g. the *r-windows* of Figure 2.8, which we walk through in detail in Section 2.3). For completeness, we also include the full collection of development set structures in Figure 2.3.



Figure 2.3: Target structures in the development set.

2.2 COLLECTING THE MINECRAFT DIALOGUE CORPUS

The Minecraft Dialogue Corpus consists of 509 human-human dialogues and game logs for the Collaborative Building Task. This section describes this corpus and our data collection process. Sample dialogues from the resulting dataset can be found in Appendix A.

2.2.1 Design of Minecraft Clients

We provided **A** with access to two Minecraft clients: the Architect client (Figure 2.4a) and the Oracle client (Figure 2.4b). Using the Architect client, **A** can fly around freely in **B**’s world and view the build region as **B** builds within it, but remains completely invisible to **B**. This client also allows **A** to interact with **B** through the Minecraft chat interface. Specifically, we have modified the Architect client such that when **A** opens the chat interface to interact with **B**, the client snaps to a third-person view behind **B** such that **A** can orient their instructions around **B**’s current perspective; once the chat window is closed, the client resumes a neutral perspective that no longer follows **B**. Separately, through the Oracle client, **A** can inspect *Target* in its entirety in a world that is inaccessible to **B**.

B was only given access to the Builder client (Figure 2.4c). Here, **B** controls a named Builder character that is embodied in the world and is tasked with building a copy of *Target* within the bounds of the build region. Like **A**, **B** has access to the Minecraft chat interface, but cannot see **A**’s location at any time (though is aware that **A** can see them and what they are looking at).

We allowed player chat to stay completely unrestricted: no language filters or spell checking systems were applied. We also modified both the Architect and Builder clients such that when either chat window was opened, signaling that either **A** or **B** intended to send a chat message, an overlay appeared on the partner’s screen indicating that the other player was typing (see Figure 2.4c). We included this to encourage players to respectfully wait their turn and avoid bombarding their partner with chat messages or **B** actions if their partner

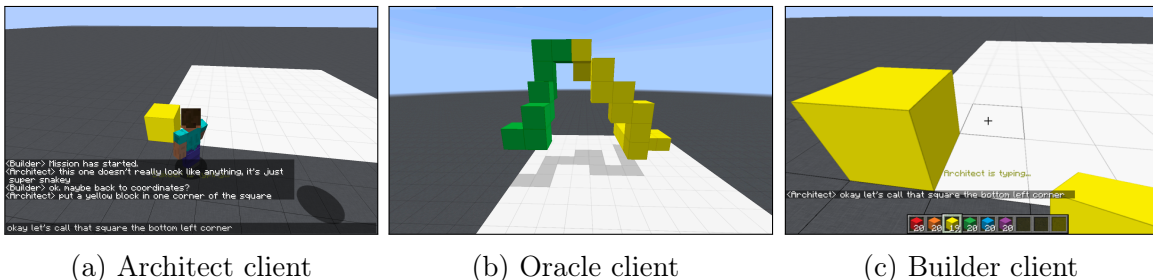


Figure 2.4: Minecraft clients for the Collaborative Building Task.

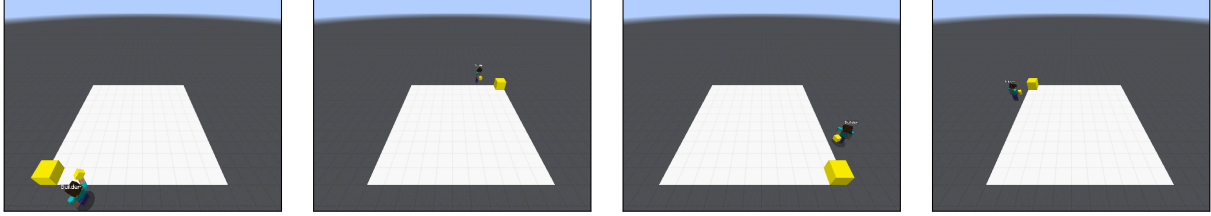


Figure 2.5: Fixed Viewer viewpoints around the build region.

was seen typing a response.

We piloted several iterations of gameplay flow using these modified Architect and Builder clients. At first, we attempted to constrain the gameplay to a strictly turn-based scenario in order to enforce clear dialogue turns by freezing each of the Architect and Builder clients in an alternating manner. However, this proved to be too cumbersome and unnatural. We eventually transitioned to a more natural gameplay environment, closer to an actual multi-player Minecraft game, where players could move and chat independently from each other. This design decision has greatly influenced our resulting data: the highly asynchronous dialogue resulted in many instances of crossed wires between players, and **A** instructions often ended up scattered across multiple utterances and interactions with **B**.

2.2.2 Data Structures and Collection Platform

We use the Microsoft Project Malmo [19] API in order to log, save, and load game states, and extend Malmo into a data collection platform by defining new functionality using the Malmo API as well as our own in-house Minecraft mods. We represent the progression of each game (involving the construction of a single target structure by an Architect and Builder pair) as a discrete sequence of game states. Although Malmo continuously monitors the game, we selectively discretize this data by only saving snapshots, or “observations,” of the game state at certain triggering moments (whenever **B** picks up or puts down a block or when either player sends a chat message). This allows us to reduce the amount of (redundant) data to be logged while preserving significant game state changes. Each observation is a JSON object that contains the following information: 1) a time stamp, 2) the chat history up until that point in time, 3) **B**’s position (a tuple of real-valued x, y, z coordinates as well as pitch and yaw angles, representing the orientation of their camera), 4) **B**’s block inventory, 5) the locations of the blocks in the build region, 6) screenshots taken from **A**’s and **B**’s perspectives. Whenever **B** manipulates a block, we also capture screenshots from four invisible “Fixed Viewer” clients hovering around the build region at fixed angles (see Figure 2.5).

2.2.3 Data Collection Procedure

Data was collected over the course of 3 weeks (approx. 62 hours overall). 40 volunteers, both undergraduate and graduate students who are native English speakers with varying levels of proficiency with Minecraft, participated in 1.5 hour sessions in which they were paired up and asked to build various predefined structures within the predefined $11 \times 9 \times 11$ sized build region. Each player was introduced to the Minecraft controls, as well as instructed on how to build floating blocks, before their first session. Builders began with an inventory of 6 colors of blocks and 20 blocks of each color. After a brief warm-up round to become familiar with the interface, participants were asked to successfully build as many structures as they could manage within this time frame. On average, each game took 8.55 minutes.

Participants were allowed to complete multiple sessions if desired; we ensured that an individual never saw the same target structure twice, and attempted as much as possible to pair them with a previously unseen partner. While some individuals indicated a preference towards either the Architect or Builder roles, roles were, for the most part, assigned in such a way that each individual who participated in repeat sessions played both roles equally often; on rare occasions, those who were completely unfamiliar with Minecraft were defaulted to the Architect role, which required less manual control of the in-game avatar and resulted in faster and smoother sessions. Each participant was assigned a unique anonymous ID across sessions.

Architects were encouraged not to overwhelm the Builder with instructions and to allow their partner a chance to respond or act before moving on. Builders were instructed not to place blocks outside the specified build region and to stay as faithful as possible to the Architect’s instructions. Both players were asked to communicate as naturally as possible while avoiding idle chit-chat.

2.3 DATA STATISTICS AND ANALYSIS

Overall statistics The Minecraft Dialogue Corpus contains 509 human-human dialogues (15,926 utterances, 113,116 tokens) and game logs for our predefined 150 target structures of varying complexity. We collected a minimum of three dialogues per structure. The training, test and development sets consist of 85 structures (281 dialogues), 39 structures (137 dialogues), and 29 structures (101 dialogues) respectively. Dialogues for the same structure are fully contained within a single split; structures in training are thus guaranteed to be unseen in test.

On average, dialogues contain 30.7 utterances: 22.5 Architect utterances (avg. length 7.9

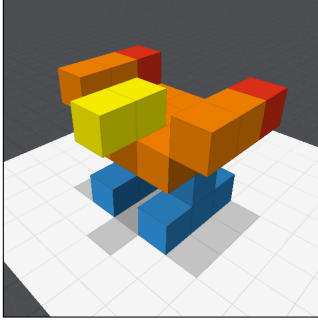


Figure 2.6: In this example, after priming **B**, **A** provides a series of instructions that are heavily dependent on **B**'s orientation. **B**'s orientation remains static throughout this exchange, but is reversed with the last instruction, where *walking forward* requires **B** to turn 180 degrees and walk backward. Without proper context, these instructions are ambiguous.

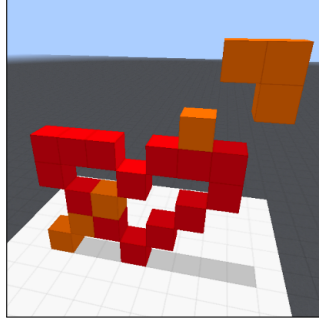
tokens), 8.2 Builder utterances (avg. length 2.9 tokens), and 49.5 Builder block movements. Dialogue length varies greatly with the complexity of the target structure (not just the number of blocks, but whether it requires floating blocks or contains recognizable substructures).

Floating blocks Blocks in Minecraft can be placed anywhere as long as they touch an existing block (or the ground). If such a supporting block is later removed, the remaining block (and any structure supported by it) will continue to “float” in place. This makes it possible to produce complex designs. Instructions for these structures varied greatly, ranging from step-by-step instructions involving temporary supporting blocks to single-shot descriptions such as, simply, “*build a floating yellow block*” (sufficient for a veteran Minecraft player, but not necessarily for a novice). General descriptions of structures also commonly referred to floating blocks or regions, e.g. “*The h will be red, consisting of a column of 4 red bricks, then one red brick floating, then two red bricks, from left to right*” or “*floating above the blue brick, we’ll make a blue C shape.*”

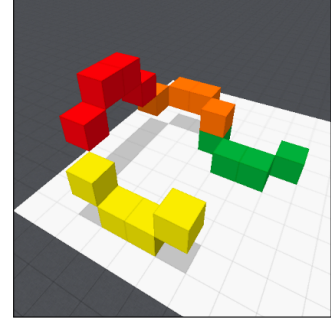
Referring expressions and ellipsis Architects made frequent use of implicit arguments and references, relying heavily on the Builder’s current perspective and their most recent actions for reference resolution. For instance, Architect instructions could include references



(a) “Either a chicken or a gun turret”



(b) “A heart that looks diseased”



(c) “A silly multicolored worm”

Figure 2.7: Architect descriptions of various target structures.

such as “two more in the same direction,” “one up,” “two towards you,” and “one right from the last thing you built.” Figure 2.6 shows an example in context, in which **A** instructs **B** to build a structure out of yellow blocks that snakes around arbitrarily. After indicating that the subsequent instructions would adhere to a static perspective, **A** issues a series of commands, each containing implicit references to **B**’s most recent actions.

Recognizable shapes and sub-structures Some target structures were designed with commonplace objects in mind. Some Architects took advantage of this in their instructions, ranging from straightforward (‘L’-shapes, “staircases”) to more eccentric descriptions (“either a chicken or a gun turret”, “a heart that looks diseased”, “a silly multicolored worm”; see Figure 2.7). To avoid slogging through block-by-block instructions, Architects frequently used such names to refer to sub-elements of the target structure. For instance, for the structure in Figure 2.7a, **A** describes the structure as a *chicken*, then continues to refer to various body parts in subsequent instructions while **B** responds in kind (see Appendix A for the full dialogue):

A: *this is either a **chicken** or a **gun turret***

...

A: *good. that’s the **feet/legs***

...

A: *now **wings**. two orange blocks in a row up and diagonal away from the **body***

...

A: *now a single red block behind each of the **wings**. still in line with the oranges*

...

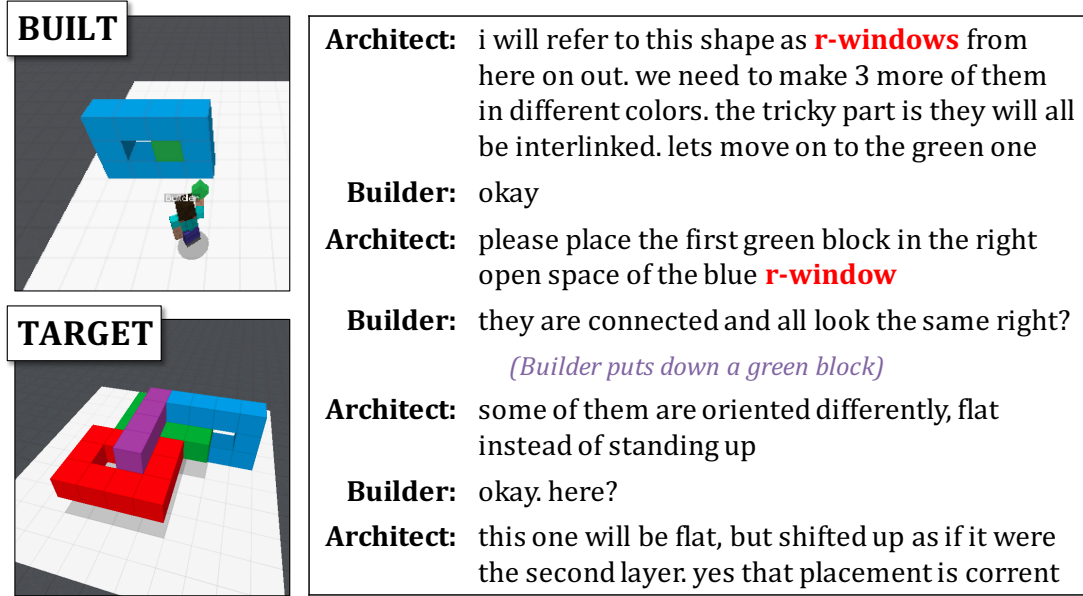


Figure 2.8: In this example, **A** defines a new term, *r-window*, for the hollow blue rectangle that **B** just built. **A** then uses the newly defined term to describe the immediate next steps in green. **A** and **B** discuss relative spatial relations and orientations of the *r-windows* without need for further explicit definitions.

A: then a row of two yellow blocks on the orange side of the **wings** for the **neck/head**
A: juts out over the edge
A: parallel to the **wings**
B: Should they be right next to the **wings**?
A: centered. one block in the middle edge orange
A: then another block coming away from the **body**

Some Architects even defined new terms, with associated properties and spatial relations, that get re-used across the dialogue. For instance, in Figure 2.8, after **A** instructs **B** how to build a rectangular substructure, **A** names the concept as an *r-window* and consistently uses it throughout the rest of the dialogue, even inspiring **B** to adopt the term ¹:

A: i will refer to this shape as **r-windows** from here on out. we need to make 3 more of them in different colors. the tricky part is they will all be interlinked. lets move on to the green one
 ...

¹It should be noted that this example is from our test set, but we felt it important to include to demonstrate the use of concepts in our data. We do not include the full dialogue in the Appendix for this reason.

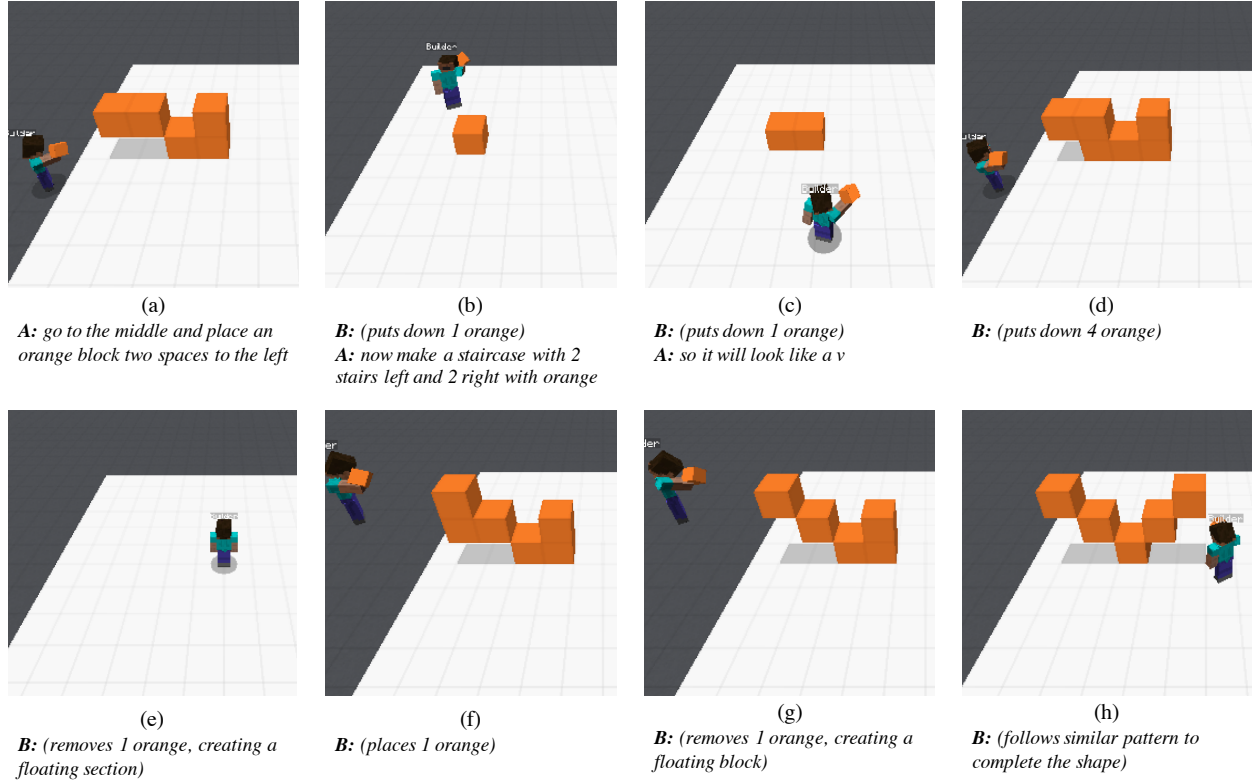


Figure 2.9: A sample sequence of human-human game states (from Jayannavar et al. (2020) [38]). The game starts with an empty grid and an initial **A** instruction (a), which **B** executes in the first action sequence (b) by placing a single block. In (c), **B** begins to execute the next **A** instruction given in (b). However, **A** interrupts **B** in (c), leading to two distinct **B** action sequences: (b)–(c) (single block placement), and (c)–(h) (multiple placements and removals).

A: next we need a purple *r-window*. it should be interlinked with the green and extending away from you. this one is not flat, much like the blue one

...

B: it might have been easier to describe the two *windows* first that are perpendicular then the interlocking one..

A: a good point builder, i will try to do better in the future. our last *r-window* is red and oriented like the green one (flat). it should be interlinked with the purple *r-window*. it should extend to the left from where you are now

Builder actions Builder action sequences are noisy, as human Builders are prone to mistakes and misunderstanding. Accidentally clicking on the Builder client may inadvertently place or remove unwanted blocks; these mistakes are commonly silently resolved by the

Builder. However, these should be distinguished from *placeholder* blocks, which are intentionally placed then removed to build floating structures. Even after multiple rounds of clarification and correction dialogues, Builders can still easily misinterpret vague Architect instructions, leading to completely incorrect block placements. Additionally, since Architects frequently interrupt, Builder action sequences are often fragmented between utterances, and reconstructing a full sequence of block placements and removals requires sifting through dialogue context. Mapping Architect instructions to their resulting interpretations in the Builder’s actions is therefore not straightforward, as shown in Figure 2.9.

Builder utterances Even though the Architect shouldered the large responsibility of describing the unseen structure, the Builder played an active role in continuing and clarifying the dialogue, especially for more complex structures. Builders regularly took initiative during the course of a dialogue in a variety of ways, including verification questions (*“is this ok?”*), clarification questions (*“is it flat?”* or *“did I clean it up correctly?”*), status updates (*“i’m out of red blocks”*), suggestions (*“feel free to give more than one direction at a time if you’re comfortable,”* *“i’ll stay in a fixed position so it’s easier to give me directions with respect to what i’m looking at”*), or extrapolation (*“I think I know what you want. Let me try,”* then continuing to build without explicit instruction).

2.4 OTHER USES OF THE CORPUS

This section describes related work that has emerged since the release of the Minecraft Dialogue Corpus that uses our dataset.

Spatial AMR: Expanded Spatial Annotation in the Context of a Grounded Minecraft Corpus (Bonn, Palmer, Cai, and Wright-Bettner, LREC 2020) [40] Drawing from the Minecraft Dialogue Corpus, this work expands the Abstract Meaning Representation (AMR) schema, adding support for fine-grained spatial semantics and object grounding, and uses the updated schema to annotate 12,600+ individual AMRs spanning 185 full Minecraft dialogues. This work contributes a large amount of structured annotation, in the form of AMRs, to our otherwise raw dataset. Though we do not investigate in this thesis how these AMRs can be leveraged for downstream tasks, future work should consider how these much richer, structured representations (and in particular, the annotations for the Builder’s frame of reference) could be used.

Learning to execute instructions in a Minecraft dialogue (Jayannavar, Narayan-Chen, and Hockenmaier, ACL 2020) [38] In this work, we define the subtask of Builder Action Prediction as the task of predicting the sequence of actions (block placements and/or removals) that a human Builder performed at a particular point in a human-human game, such as those of Figure 2.9. Our models process the game history along with a 3D representation of the evolving world to predict actions in a sequence-to-sequence fashion, and we show that the models, especially when conditioned on a suitable amount of game history and trained on larger amounts of synthetically generated data, improve over naive baselines. Some of the outcomes and lessons learned from designing the Builder have influenced design decisions and experiments for the Architect, which we describe in later chapters.

2.5 CONCLUSION

In this chapter, we introduced the Minecraft Dialogue Corpus of game logs involving humans completing the Collaborative Building Task. In order to understand and generate the type of natural, asynchronous dialogue that arises, both **A** and **B** must be able to account for dynamic **B** perspectives and ground both low- and high-level concepts in a constantly-changing world state. Because the roles are asymmetric, **A** additionally must be able to compare *Built* and *Target* structures to properly guide **B** towards the goal, correcting them along the way; meanwhile, **B** needs to be robust to **A** interruptions while executing sequences of block actions, asking appropriate questions when needed and learning concepts on-the-fly. Humans and agents alike must integrate several challenging skills in order to be successful at this task, and we have demonstrated this by highlighting examples from our dataset.

In the next chapter, we take a first stab at building agents to play the Architect role. We focus on an important subtask, Architect utterance generation, and discuss how we process the raw game state representations described in this chapter into a meaningful representation for utterance generation models.

Chapter 3: Architect Utterance Generation

In the previous chapter, we introduced the Minecraft Dialogue Corpus, a collection of game logs of humans successfully completing the Minecraft Collaborative Building Task, and demonstrated various challenging aspects of the task. We now turn our focus to building interactive agents that can perform this task. Although the Minecraft Dialogue Corpus was motivated by our ultimate goal of building agents that can successfully play an entire collaborative building game as Architect or Builder, we first consider the task of **Architect utterance generation**: given access to the entire game state context leading up to a certain point in a human-human game at which the human Architect spoke next, we aim to generate a suitable Architect utterance.

Architect utterance generation is a much simpler task than developing a fully interactive Architect or Builder, but it still captures some of the essential difficulties of the Architect’s role. Architects need the ability to understand the preceding dialogue and compare the built structure against the target structure in order to be able to give instructions appropriately and correctly, correct Builders’ mistakes and answer their questions. Models that generate Architect utterances are also much easier to evaluate than full interactive systems: not only can we use automatic utterance-level metrics such as BLEU [41], but we can also run human evaluations that are relatively cheaper and faster than full interactive evaluations. We also aim to leverage models developed for this task to at least bootstrap a fully interactive Architect (which will also need to decide when to speak, as well as deal with potentially much noisier dialogue histories than those we are considering here), and explore this more in Chapter 6.

This chapter is structured as follows: we begin with a technical background on models for utterance generation that have inspired our models in Section 3.1. We formally define the Architect utterance generation task in Section 3.2. We then introduce the baseline sequence-to-sequence Architect utterance model [39] in Section 3.3. We motivate and describe how we augment the baseline model with simple world state representations in Section 3.4. We describe our experiments in Section 3.5. After proposing and motivating automated metrics and human evaluation criteria for this task, we evaluate and analyze our model outputs using automated metrics, qualitative analysis and human evaluation in Section 3.6. Finally, we describe and evaluate additional improvements to the aforementioned model in Sections 3.7 and 3.8.

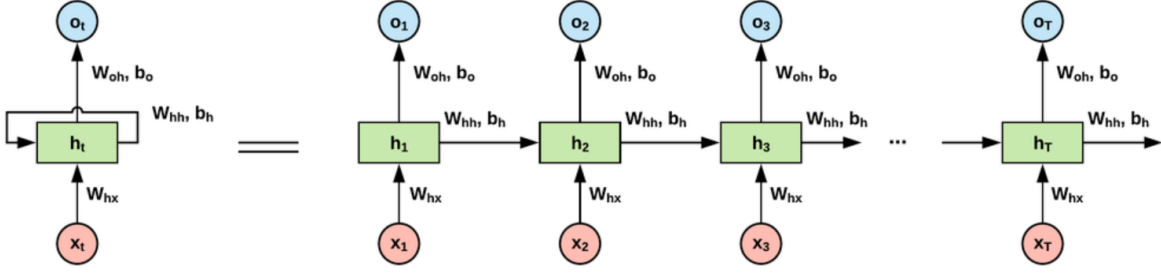


Figure 3.1: A vanilla RNN cell (left) and its computation graph when unfolded (right) [45].

3.1 BACKGROUND

Recurrent neural networks (RNNs) [42, 43] are neural networks designed to process sequential data. Specifically, RNNs operate on sequences x_1, \dots, x_T , where each input vector x_t has a time step index t , by sharing parameters across several time steps using a recurrent formulation. Using this formulation, RNNs retain a memory of previous outputs and can handle and produce sequences of variable length. Even further, bidirectional RNNs [44] combine information about past (backward) and future (forward) outputs simultaneously by connecting the hidden layers of two RNNs in opposite directions to the same output. RNNs are thus a popular method of processing text in various NLP tasks.

A vanilla RNN cell works by iteratively updating a hidden state h . At every time step t , the next hidden state h_t is updated with respect to the previous hidden state h_{t-1} and the next input x_t , which is then used to calculate the next output o_t . o_t can then be normalized to a vector of probabilities y_t for discrete outputs (such as tokens or characters) using the softmax operator. The RNN is defined by the following equations:

$$\begin{aligned} h_t &= \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \\ o_t &= W_{ho}h_t + b_o \\ y_t &= \text{softmax}(o_t) \end{aligned} \tag{3.1}$$

where the matrices W_{xh} , W_{hh} and W_{ho} as well as the bias vectors b_h and b_o are learned parameters shared across time steps t . To compute the gradient in an RNN, the computational graph is unrolled across the T time steps and standard backpropagation algorithm is applied. This process is called backpropagation through time (BPTT).

In text data, the input tokens x_1, \dots, x_T are first represented as one-hot vectors. That is, with an input vocabulary of size N , tokens are mapped to vectors of max length N , where the n th bit of each vector indicates the presence of the n th token in the vocabulary. Then,

learned word embeddings are used to map these representations down into a smaller space such that semantically similar tokens have similar representations. Instead of learning these embeddings on-the-fly, a common approach is to instead use pretrained word embeddings, such as GloVe vectors [46], which apply unsupervised learning algorithms to large amounts of text to learn these semantic similarities. These word embeddings, rather than the one-hot token representations, constitute the actual inputs to the RNN.

Vanilla RNNs have been known to suffer from the *vanishing gradient problem*, in which the gradient is reduced to 0 after multiple time steps, and as a result have trouble learning long-range dependencies. To combat this problem, two popular variants of the RNN cell have emerged: Long Short-Term Memory units (LSTMs) [47] and Gated Recurrent Units (GRUs) [48]. Both variants make use of internal gating mechanisms to combat the vanishing gradient problem and have been shown to exhibit comparable performance. Since our models in this thesis make use of GRUs, we describe the GRU architecture here.¹

A GRU unit makes use of a reset gate r and an update gate z . At every time step, the reset gate defines how to combine the new input with previous memory, while the update gate determines how much of the previous memory should be retained. Intuitively, these gates regulate the flow of information by learning which data in a sequence is important to retain or discard, making it easier to transmit relevant information over a longer series of time steps. Specifically, a GRU is defined using the following equations:

$$\begin{aligned} z_t &= \sigma(W^{(z)}x_t + U^{(z)}h_{t-1}) \\ r_t &= \sigma(W^{(r)}x_t + U^{(r)}h_{t-1}) \\ h'_t &= \tanh(Wx_t + r_t \odot Uh_{t-1}) \\ h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot h'_t \end{aligned} \tag{3.2}$$

where $W^{(z)}, U^{(z)}, W^{(r)}, U^{(r)}, W$, and U are learned weight matrices and \odot is an element-wise multiplication.

A recent alternative to the RNN is the transformer [49], a powerful, parallelizable neural architecture that makes use of a self-attention mechanism to process sequences. However, as these architectures became popular around the same time as when we developed our models, we do not explore using them in this thesis, and leave this extension to future work.

¹GRUs are better suited to training on smaller datasets as they have relatively fewer parameters. We experimented with both LSTMs and GRUs for our models, but found that GRUs performed better, possibly due to the limited amount of training data we have.

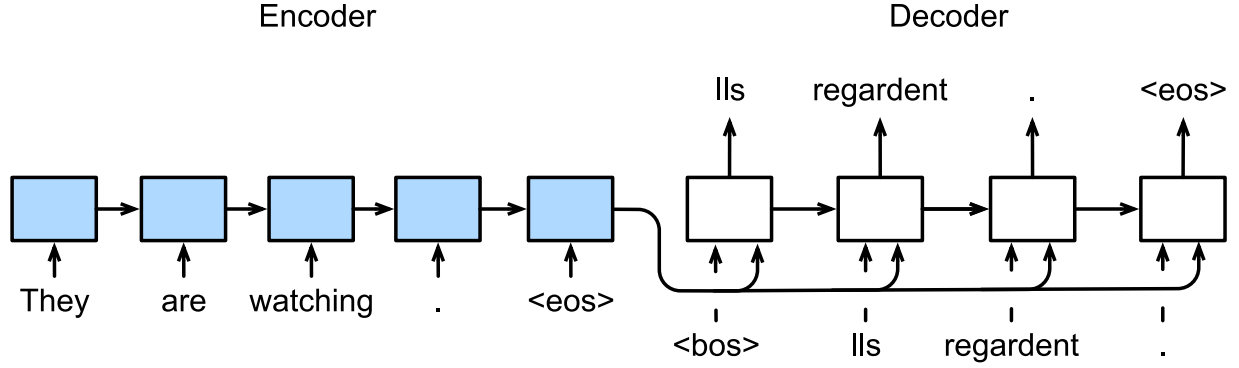


Figure 3.2: Encoder-decoder (sequence-to-sequence) architecture [53].

Sequence-to-sequence models [50, 51, 52] are neural architectures that use RNNs to map input sequences $X = x_1, \dots, x_{n_x}$ to output sequences $Y = y_1, \dots, y_{n_y}$, which are not necessarily of the same length, that are commonly used for natural language generation (NLG). Originally used for machine translation tasks, the sequence-to-sequence model uses at its core an encoder-decoder architecture (Figure 3.2). First, an encoder RNN processes the input sequence X and produces a fixed-length context C , usually as a function of the RNN’s final hidden state. Then, a decoder RNN is conditioned on C to generate the output sequence Y .

The encoder and decoder RNNs are trained jointly to maximize the average log likelihood $\log P(y_1, \dots, y_{n_y} | x_1, \dots, x_{n_x})$ over all pairs of sequences x and y in the training set. This is commonly implemented by minimizing cross-entropy loss with respect to the tokens in the ground truth output sequence. Specifically, the decoder RNN produces probability vectors p_t over the output vocabulary at every time step t , which are used to compute the probability of a given target sequence as the product of the probabilities of each token:

$$P(y_1, \dots, y_{n_y}) = \prod_{t=1}^{n_y} p_t[y_t] \quad (3.3)$$

The probability of the target sequence is maximized by minimizing its negative log likelihood:

$$\begin{aligned} -\log P(y_1, \dots, y_{n_y}) &= -\log \prod_{t=1}^{n_y} p_t[y_t] \\ &= -\sum_{t=1}^{n_y} \log p_t[y_t] \end{aligned} \quad (3.4)$$

which is equivalent to minimizing the cross-entropy between the target and predicted distri-

butions.

At test time, there are two main methods to decode output sequences using a sequence-to-sequence network. The most straightforward method is greedy decoding: at every decoder time step, the most likely predicted word from the previous time step is fed in as input. However, a greedily-sampled output sequence may be far from optimal. Therefore, another popular approach is to use beam search decoding. In beam search, we keep track of a *beam* of k hypotheses. At every time step, each of the k hypotheses is expanded; afterwards, the top k expanded hypotheses are retained. As a result, a top- k list of k best overall hypotheses are retained. Recently, improvements to regular beam search have been made, such as sampling tokens from a top- p list defined by the decoder’s predicted output distributions [54] and promoting diverse outputs by penalizing sibling hypotheses within the beam [55].

Combined with additional architecture, training regimes and/or decoding schemes, sequence-to-sequence models have enjoyed success in a variety of NLG tasks, including meaning representation-to-text generation [56] and abstractive text summarization [57]. Furthermore, RNN decoders have been shown to produce meaningful output sequences when conditioned on multimodal inputs, such as on image representations, as in the case of image captioning [58, 59].

Evaluation While human evaluation of NLG outputs is a gold standard, it is often not feasible to run such expensive evaluations on a large number of outputs. This is especially true in the Minecraft Collaborative Building Task, where a thorough evaluation requires processing the world state changes in the context of a real game. Therefore, to compare models quantitatively, we use BLEU (bilingual evaluation understudy) [41], a popular corpus-level modified n -gram precision metric borrowed from machine translation literature. BLEU measures the n -gram overlap between the generated output and ground truth references with a brevity penalty for unreasonably short outputs.

To compute BLEU for n -grams of order i , a modified i -gram precision is first computed by counting the number of i -grams in the generated output that also occur in the references, denoted $Matched(i)$, and dividing by the total number of i -grams in the generated output, denoted $H(i)$:

$$P(i) = \frac{Matched(i)}{H(i)} \quad (3.5)$$

where

$$Matched(i) = \sum_{t_i} \min\{C_g(t_i), \max_j C_{gj}(t_i)\} \quad (3.6)$$

where t_i is an i -gram in the generated output g , $C_h(t_i)$ is the count of occurrences of t_i in

g , and $C_{gj}(t_i)$ is the count of occurrences of t_i in ground truth reference j . Essentially, this limits the number of matches for a particular i -gram to the maximum number of times it appears in any given reference, such that randomly generating long output sequences with multiple spurious i -gram matches does not increase the generated output’s BLEU score.

Additionally, a brevity penalty ρ is used to penalize short outputs:

$$\rho = \exp\{\min(0, \frac{n-L}{n})\} \quad (3.7)$$

where n is the length of the generated output and L is the length of the ground truth reference. In cases of multiple references, L is commonly taken to be either the length of the shortest reference, or the length of the reference with length closest to that of the generated output.

The final BLEU score calculation is a function of modified precision of i -grams up to order N with the added brevity penalty:

$$\text{BLEU} = \rho \cdot \left\{ \prod_{i=1}^N P(i) \right\}^{1/N} \quad (3.8)$$

Unfortunately, BLEU has been shown to have poor correlation with human judgments [60, 61], especially in NLG and dialogue response generation tasks [25, 62]. BLEU was also designed to be used with multiple ground truth reference sentences for every generated output, which are not available to us without additional paraphrasing effort; and even then, multiple semantically different candidate responses can be valid at a given point in time. To avoid relying on BLEU for model evaluation, we perform human evaluation and qualitative analysis of our outputs whenever possible. Finally, BLEU is not designed to estimate a measure of task success, and may rate a well-worded yet incorrect instruction highly. To this end, we design additional term-based automated metrics, discussed in Section 3.6.1.

3.2 TASK DEFINITION

We define the Architect utterance generation task as follows: given access to the entire game state context leading up to a certain point in a human-human game at which the human Architect **A** spoke next, we aim to generate a suitable **A** utterance. The task does not require that we model *when* **A** should speak. Instead, we generate a data sample for every point in a game in which the human **A** spoke next; every sample includes the target structure *Target*, the full history of utterances and actions between **A** and **B**, and the

partially built structure *Built*, up to that point.

As we have seen from human performance on the Collaborative Building Task in the Minecraft Dialogue Corpus, in order to be successful at this task, **A** must be able to:

- **Describe *Target* and its substructures** beyond using instructions consisting only of simple *blocks*; **A** should ideally be able to describe a structure in terms of *rows*, *columns*, *diagonals*, *staircases*, etc. as well as higher-level, possibly more subjective terms such as *giraffes*, *tables*, or *cursive ok*
- **Align *Target* and *Built*** in order to compare them and determine next steps, since *Built* can be a rotated and translated version of *Target* within the build region
- **Provide clear instructions** which, in order to be easily grounded by **B**, should appropriately make use of **B**'s perspective and block actions
- **Identify mistakes and generate corrections** since task success relies heavily on **A**'s ability to correct **B** if they go off-course
- **Answer **B**'s questions appropriately** when **B** requests clarification or verification

Because we did not enforce a strict turn-based structure, **A** instructions can be fragmented across a sequence of utterances, each utterance adding more clarification to the last:

A: *now we'll do yellow*

A: *it'll make a diagonal connectin the top orange to the right-most red*

A: *3 yellow in total*

On the other hand, they can also be multiple sentences long, embodying multiple ideas:

A: *hello builder, i will tell you this. it appears we are creating a belltower. but first i will start with step by step instructions. we will start with green blocks*

A: *please start with 8 green blocks extending straight up. this will need to be placed as far right as you can, preferably centered along the other axis*

The task treats each of the above utterances as individual samples in our data, and we have not made any efforts to segment or merge these utterances to produce cleaner instruction boundaries.

Finally, we assume perfect information about the world (*Built* and *Target*) represented as exact coordinates. One could imagine a scenario in which we must rely on an added visual component to build a representation of blocks in the world via screenshots from the Architect's point of view. However, this introduces numerous additional vision-based

problems (e.g. extrapolating beyond field-of-view, reconstructing a 3D scene, dealing with obstructions, etc.) which are outside the scope of this thesis.

In the following sections, we discuss how we represent the given dialogue history and world state as inputs to an utterance generation model.

3.3 SEQ2SEQ ARCHITECT UTTERANCE MODEL

We define a sequence of models for Architect utterance generation. This section introduces our most basic variant, a sequence-to-sequence model [50] that conditions the next utterance on the preceding dialogue. Since Architects need to compare the current state of the build region against the target structure, we augment this model in the next section with world state information.

Dialogue History Encoder We encode the entire dialogue history as a sequence of tokens in which each player’s utterances are contained within speaker-specific start and end tokens, e.g.:

```
<A> hello again builder </A> <B> hello </B> <A> we will start using green
blocks </A> <A> i need you to build 4 green blocks straight from where you
are standing , all on the ground </A> <B> like this ? </B>
```

Each utterance corresponds to a single chat message, and may consist of multiple sentences. These tokens are fed through a word embedding layer and subsequently passed through a bidirectional RNN [44] to produce an embedding of the entire dialogue history in the encoder RNN’s final hidden state.

Output Utterance Decoder The output utterance is generated by a decoder RNN conditioned on the discourse context. In standard fashion, the final hidden state of the encoder RNN is used to initialize the hidden state of the decoder RNN.

3.4 SIMPLE WORLD STATE REPRESENTATIONS: BLOCK COUNTERS

To be able to give accurate instructions, the Architect requires a mental model of how the target structure can be constructed successfully given the current state of the built structure. Since the Builder’s world is not explicitly aligned to the target structure (our space does not contain any markers that would indicate cardinal directions or other landmarks, and we consider any built structure a success as long as it matches the target structure

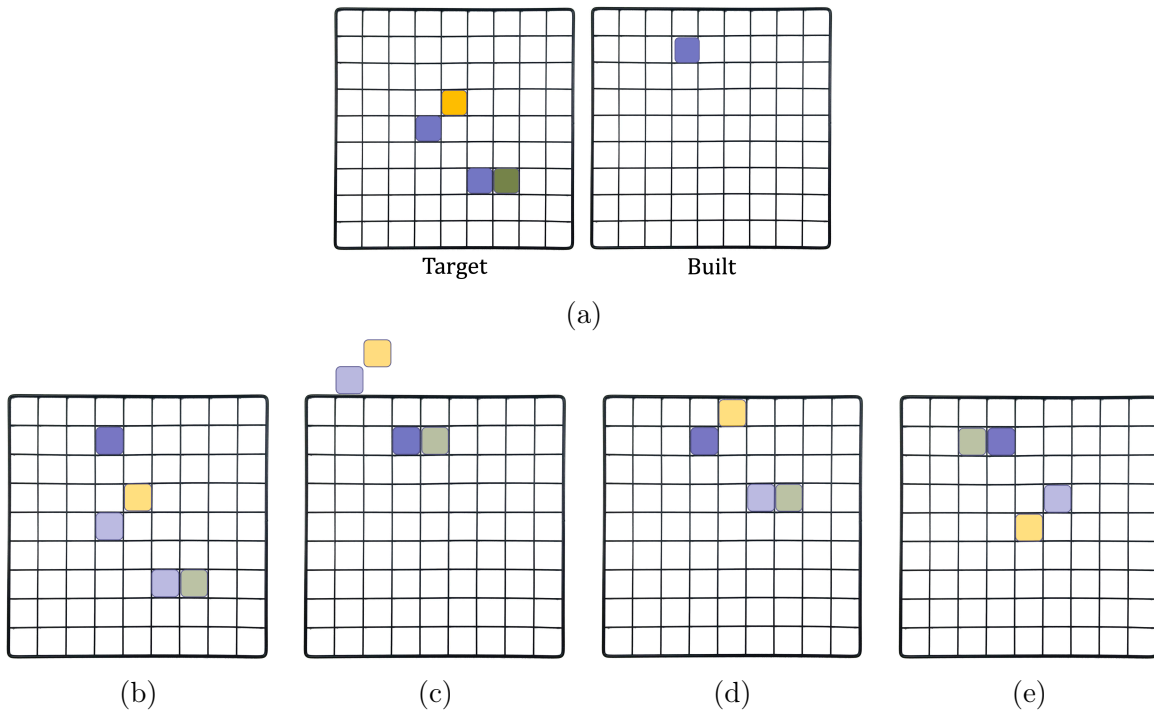


Figure 3.3: Toy 2D example aligning *Target* and *Built* (a). Alignment (b) is suboptimal, while (c) is invalid, as it lies outside the build region. Alignments (d) and (e) are both optimal and valid.

and fits completely into the Builder’s build region), this model must consider all possible translational and rotational alignment variants, although we assume it can ignore any sub-optimal alignments. For any given alignment, we compute the Hamming distance between the built structure and the target (the total number of blocks of each color to be placed and removed), and only retain those alignments that have the smallest distance to the target. Once the game has progressed sufficiently far, there is often only one optimal alignment between built and target structures, but in the early stages, a number of different optimal alignments may be possible. Our world state representation captures this uncertainty.

3.4.1 Computing Block Counters

To visualize this idea of optimal alignments, we refer to Figure 3.3, which depicts the alignment problem in a simplified 2D case. Given *Target* and *Built* in (a), there are multiple ways we can align the two structures. Simply superimposing *Target* onto *Built* produces a valid alignment (b), but it is clearly suboptimal, as it would involve removing the existing blue block and starting from scratch. There are also a number of invalid ways that *Target* and *Built* can be aligned that result in blocks outside the build region, e.g. (c). By

translating and rotating *Target*, we can achieve optimal alignments (d) and (e), in which *Built* can be transformed into *Target* with the fewest number of block movements to the goal. Because these alignments are equally valid, we should represent this uncertainty over optimal alignments when building a world state representation for our models.

Extending to the 3D case, Figure 3.4 depicts a target structure (left) and a point in the game at which two adjacent red blocks have been placed (right). In this scenario, a mistake has already happened, as the target instead requires two red blocks with a gap. There are multiple ways these two blocks can be aligned to the legs such that the rest of the structure can be built, and all of these optimal alignments require that one of these erroneous red blocks be removed. We can identify three potential paths (left, up, and down) to continue the structure by extending it along the four cardinal directions. A permissibility check disqualifies the option of extending to the right, as blocks would end up placed outside the build region. These remaining paths, considered equally likely, indicate the colors and locations of blocks to be placed and removed. A summary of this information forms the basis of the input to our model.

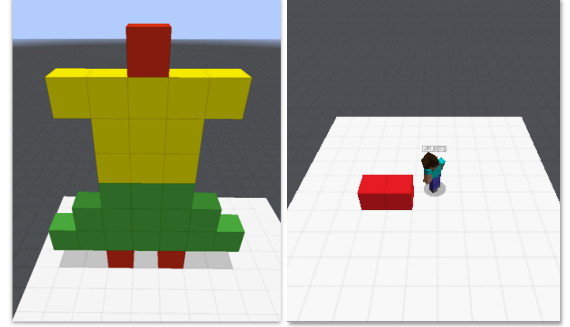


Figure 3.4: A target structure (left) and corresponding built structure at a certain point in the game (right).

Computing the distance between structures Computing the Hamming distance between the built and target structure under a given alignment tells us also which blocks need to be placed or removed. A structure S is a set of blocks (c, x, y, z) . Each block has a color c and occupies a location (x, y, z) in absolute coordinate space (i.e., the coordinate system defined by the Minecraft client). A structure’s position and orientation can be mutated by an alignment A in which S undergoes a translation A_T (shift) followed by a rotation A_R , denoted $A(S) = A_R(A_T(S))$. We only consider rotations about the vertical axis in 90-degree intervals, but allow all possible translations along the horizontal plane. The symmetric difference between the target T and a built structure S w.r.t. an alignment A , $\text{diff}(T, S, A)$, consists of the set of blocks to be placed, $B_p = A(T) - S$ and the set of blocks to be removed from S , $B_r = S - A(T)$.

$$\text{diff}(T, S, A) = B_p \cup B_r \quad (3.9)$$

The cardinality $|\text{diff}(T, S, A)|$ is the Hamming distance between $A(T)$ and S .

Feasible next placements Architects’ instructions often concern the immediate next blocks to be placed. Since new blocks can only be feasibly placed if one of their faces touches the ground or another block, we also wish to capture which blocks B_n can be placed in the immediate next action. B_n , the set of blocks that can be feasibly placed, is a subset of B_p .

It is important to note here that any floating blocks that require temporary placeholder blocks are not deemed feasible unless their corresponding placeholder block has already been placed. Additionally, the placeholder blocks themselves are not counted towards the feasible block total, as they would result in a suboptimal alignment of the built and target structures. Future models must overcome this shortcoming given the large number of floating blocks in our structures.

Block counters To obtain a summary representation of the optimal alignments (without detailed spatial information), we represent each of the sets B_p and B_r (as well as B_n) of an alignment $A = B_p \cup B_r$ as sets of counters over block colors (indicating how many blocks of each color remain to be placed [next] and to be removed). We compute the set of expected block counters for each color $c \in \{\text{red, blue, orange, purple, yellow, green}\}$ and action $a \in \{p, r, n\}$ as the average over all k optimal alignments $A^* = \arg \min_A (|\text{diff}(T, S, A)|)$.

$$E[\text{count}_{c,a}] = \frac{1}{k} \sum_{i=1}^k \text{count}_{c,a}^i \quad (3.10)$$

With six colors, and three sets of blocks (all placements, next placements, removals), we obtain an 18-dimensional vector of expected block counts.

3.4.2 Block Counter Models

We augment our basic seq2seq model with two variants of block counters that capture the current state of the built structure:

Global block counters are 18-dimensional vectors (capturing expected overall placements, next placements, and removals for each of the six colors) that are computed over the whole build region.

Local block counters Since many Builder actions involve locations immediately adjacent to their last action, we construct local block counters that focus on and encode spatial information of this concentrated region. Here, we consider a $3 \times 3 \times 3$ cube of block locations:

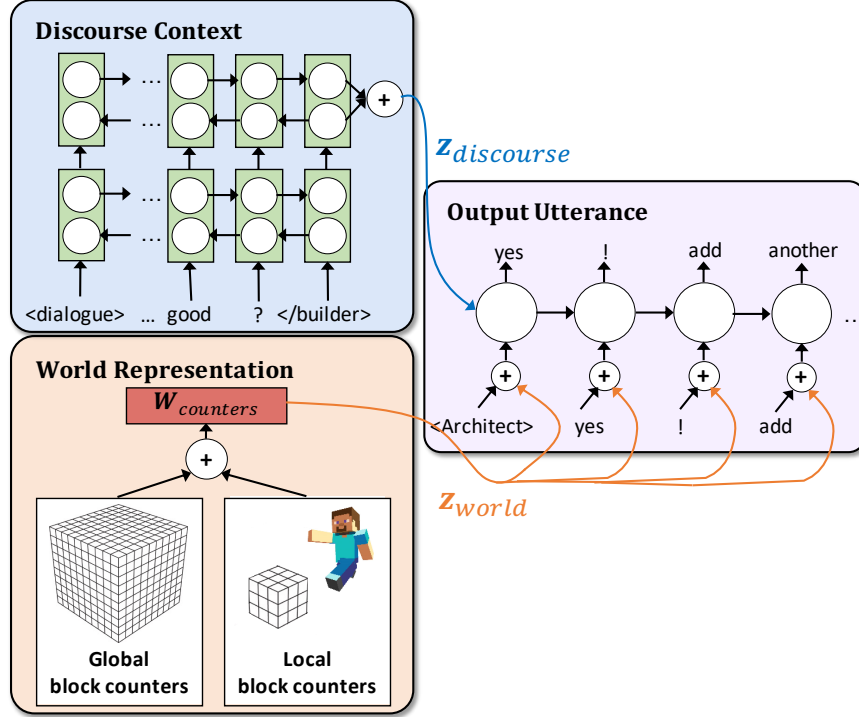


Figure 3.5: An overview of the full model combining global and local world representation variants.

those directly surrounding the location of the last Builder action as well as the last action itself. We compute a separate set of block counters for each of these 27 locations. Using the Builder’s position and gaze, we deterministically assign a relative direction for each location that indicates its position relative to the last action in the Builder’s perspective, e.g., “left”, “top”, “back-right,” etc. The 27 18-dimensional block counters of each location are concatenated, using a fixed canonical ordering of the assigned directions.

Adding block counters to the model To add block counters to our models, we found the best results by feeding the concatenated global and local counter vectors through a single fully-connected layer before concatenating them to the word embedding vector that is fed into the decoder at each time step (Figure 3.5).

3.5 EXPERIMENTAL SETUP

Data Our training, test and dev splits contain 6,548, 2,855, and 2,251 Architect utterances.

Training We trained for a maximum of 40 epochs using the Adam optimizer [63] with a learning rate of 0.0001. During training, we minimize the sum of the cross-entropy losses between each predicted and ground truth token. We stop training early when perplexity on the held-out development set had increased monotonically for two epochs.

Decoding We use beam search decoding to generate the utterance with the maximum log-likelihood score according to our model normalized by utterance length (beam size = 10). In order to promote diversity of generated utterances, we use a γ penalty [55] of $\gamma = 0.8$. These parameters were found by a grid search on the development set for our best model.

Hyperparameters We use 300-dimensional Gated Recurrent Units (GRUs) [48] for all RNN modules and use 300-dimensional pretrained GloVe word embeddings [46]. All linear layers were initialized using Xavier initialization [64]. All RNNs have a hidden state size of 300. In cases where we use a bidirectional encoder RNN, the sum of the two final hidden states in either direction constitutes the final encoding of dialogue history (used to initialize the decoder). See Appendix D.1 for detailed hyperparameter configurations.

3.6 RESULTS AND ANALYSIS

We evaluate our models in three ways: we use automated metrics to assess how closely the generated utterances match the human utterances. For a random sample of 100 utterances per model, we use human evaluators to identify dialogue acts and to evaluate whether the generated utterances are correct in the given game context. Finally, we perform a qualitative analysis of our best model.

3.6.1 Automated Evaluation

BLEU To evaluate how closely the generated utterances resemble the human utterances, we report standard BLEU scores [41]. For a longer discussion on using BLEU as an automated metric for utterance generation tasks, see Section 3.1.

Term-specific precision and recall Since our goal is to use the Architect utterance model as a stepping stone to a fully interactive Architect in a task-oriented dialogue setting, we also wish to judge model performance by approximating a measure of overall task success in an automatic fashion. A successful Architect must be able to, minimally, describe instructions using the correct colors, spatial relations, and actions (placements vs. removals), as

well as provide appropriate feedback when asked yes/no questions by the Builder; mentioning the wrong color or block location is sure to draw the Builder further away from the goal.

To this end, we also compute (modified) precision and recall of a number of lists of domain-specific keywords that are instrumental to task success: colors, spatial relations, and other words that are highly indicative of dialogue acts (e.g., responding “yes” vs. “no”, instructing to “place” vs. “remove”, etc.). These lists also capture synonyms that are common in our data (e.g. “yes”/“yeah”), and were obtained by curating non-overlapping lists of words (with a frequency ≥ 10 across all data splits) that are appropriate to each category. These word lists are in Appendix B and synonym lists are in Appendix C.

We report precision and recall scores per category, and for an “all keywords” list consisting of the union of all category word lists. For each category, we reduce both human and generated utterances to those tokens that occur in the corresponding keyword list: “*place another red left of the green*” reduces to “*red green*” for color, to “*left*” for spatial relations and “*place*” for dialogue.

For a given (reduced) generated sentence S_g and its associated (reduced) human utterance S_h , we calculate term-specific precision (and recall) as follows. Any token t_g in S_g matches a token t_h in S_h if t_g and t_h are identical or synonyms. Similar to BLEU’s modified unigram precision, once t_g is matched to one token t_h , it cannot be used for further matches to other tokens within S_h . Counts are accumulated over the entire corpus to compute the ratio of matched to total tokens in S_g (or S_h).

Ablation study Table 3.1 shows the results of an ablation study on the development set. All model variants here share the same RNN parameters. While the individual addition of global and local block counters each see a slight boost in performance in precision and recall respectively, combining them as in our final model shows significant performance increase, especially on colors.

Test set results We finetune our most basic and most complex model via a grid search over all architectural parameters and dropout values on the development set. The best model’s results on the test set are shown in Table 3.2. Our full model shows noticeable improvements on each of our metrics over the baseline. Most promising is again the significant increase in performance on colors, indicating that the block counters capture necessary information about next Builder actions.

Model	BLEU				Precision / Recall				spatial	dialogue
	B-1	B-2	B-3	B-4	all keywords	colors				
seq2seq	14.9	6.9	3.8	2.1	12.0 / 10.3	8.4 / 12.1			9.9 / 9.1	16.5 / 19.1
+ global	16.1	7.7	4.1	2.4	12.9 / 11.6	14.4 / 15.5			8.8 / 7.0	19.1 / 18.8
+ local	16.0	7.9	4.5	2.6	13.5 / 13.8	13.3 / 23.5			9.5 / 11.3	19.3 / 22.0
+ both	16.2	8.1	4.7	2.8	14.5 / 13.8	14.8 / 23.3			10.7 / 9.5	17.9 / 20.6

Table 3.1: BLEU and term-specific precision and recall scores for the ablation study on the development set.

Model	BLEU				Precision / Recall				spatial	dialogue
	B-1	B-2	B-3	B-4	all keywords	colors				
seq2seq	15.3	7.8	4.5	2.8	11.8 / 11.1	8.1 / 17.0			9.3 / 8.6	17.9 / 19.3
block counters	15.7	8.1	4.8	2.9	13.5 / 14.4	14.9 / 28.7			8.7 / 8.7	18.5 / 19.9

Table 3.2: BLEU and term-specific precision and recall scores of the seq2seq and the full model on the test set.

3.6.2 Human Evaluation

As we have seen, automated metrics, and especially BLEU, do not necessarily capture measures of generation quality and task success very well. In order to better evaluate the quality of generated utterances as well as benchmark human performance, we performed a small-scale human evaluation of Architect utterances. We asked 3 human participants who had previously completed the Minecraft Collaborative Building Task to evaluate 100 randomly sampled scenarios from the test set. Each scenario was reenacted in a Minecraft Architect client from an actual human-human game such that evaluators could inspect the build region and see the Builder in the same fashion as an Architect in a real game. Using this client, we simulated a window of context of dialogue and Builder actions (consisting of at least the last seven Builder’s and Architect’s actions, but always including the previous Architect’s utterance). Then, we presented 3 candidate Architect utterances to follow that context (one each generated from the models in Table 3.2 as well as the original human utterance) to the evaluators in randomized order.

For each scenario, we first asked evaluators to judge the utterances on fluency, dialogue acts, appropriateness and executability (defined below) without giving them access to the target structure in order to evaluate our models’ general ability to generate utterances. Afterwards, we asked them to evaluate utterances for correctness with respect to the target structure by revealing *Target* to them in an Oracle client. For each of the human evaluation

Model	Perfectly fluent	Somewhat fluent/disfluent	Completely disfluent
seq2seq	97.0	3.0	0.0
+ global & local	93.0	5.0	2.0
human	83.0	17.0	0.0

Table 3.3: Percentage of utterances deemed fluent by majority vote across 3 human evaluators.

criteria, we include here full descriptions of the evaluation guidelines as well as evaluation results and inter-rater reliability metrics of human judgments using Krippendorff’s alpha [65].

Fluency Evaluators were asked to rate the fluency of an utterance by selecting one of the following categories:

- *Perfectly fluent*: the utterance contains no spelling or grammatical mistakes and is well-formed in the context of English text chat. The utterance may not necessarily consist of complete sentences, but consists of long enough sentences to remain reasonably grammatical given the dialogue context.
- *Somewhat fluent/disfluent*: the utterance contains mistakes but still contains parts that resemble fluent English chat. Mistake types can include: 1) typos, 2) inappropriate use or addition of punctuation, 3) run-on sentences, unnecessary repetition, 4) inappropriately dropped words, etc.
- *Completely disfluent*: the utterance is word salad.

Fluency results are shown in Table 3.3 ($\alpha = 0.774$). While models are trained to produce mostly syntactically mistake-free utterances, humans are prone to producing utterances with typos and sentence fragments in a text chat.

Dialogue acts Evaluators were asked to choose all dialogue acts from a predefined set that categorized a candidate utterance. The predefined categories, determined after a manual qualitative analysis of utterances in the development set, are as follows:

- *Instruct **B***: the utterance instructs **B** to move, place or remove blocks, or otherwise execute some action within the game using their player character (“Place a red block”, “Move around to the left corner”) ($\alpha = 0.884$)

Model	Instruct B	Describe Target	Answer question	Confirm B 's actions/plans	Correct/ clarify A/B	Other
seq2seq	76.0	12.0	7.0	9.0	3.0	4.0
+ global & local	72.0	14.0	8.0	9.0	3.0	4.0
human	47.0	14.0	12.0	17.0	23.0	8.0

Table 3.4: Percentage of utterances categorized as a given dialogue act. Labels were determined per dialogue act by majority vote across three human evaluators. An utterance can belong to multiple dialogue acts.

- *Describe Target*: the utterance provides a description of the target structure or elements of it, such as blocks and substructures within the target (“*We’re going to build a 3x3*”, “*Next we’ll do wings*”) ($\alpha = 0.713$)
- *Answer question*: the utterance provides a response to a question posed by **B** (“*3 high*” in response to “*How tall?*”, “*perfect!*” in response to “*Is this right?*”) ($\alpha = 0.802$)
- *Confirm **B**’s actions or plans*: the utterance provides a confirmation (“*yes*”, “*that’s right!*”, etc.) or rejection (“*no*”, “*sorry*”, etc.) in response to actions that **B** has taken or plans proposed/executed by **B** ($\alpha = 0.696$)
- *Correct or clarify **A** or **B***:² the utterance rectifies mistakes made by **A** or **B** or provides additional clarifying information (“*No, get rid of the last block you placed*”, “*one more over to the left*”) ($\alpha = 0.778$)
- *Other*: other utterance types not covered by the above categories, including but not limited to: offhand comments, chitchat, greetings, etc. (“*Hello Builder*”, “*Haha, I couldn’t see that side*”) ($\alpha = 0.804$)

An utterance could belong to any number of categories; e.g., “*great! now place a red block*” is both a confirmation as well as an instruction.

Results can be found in Table 3.4. These results show a significantly higher diversity of utterance types generated by humans. Humans provided instructions only about half of the time, and devoted more energy to providing higher-level descriptions of the target, responding to the Builder’s actions and queries, and rectifying mistakes. On the other hand,

²This category was originally split into two separate but very similar categories, “*Correct **B**’s actions or plans*” and “*Clarify or correct **A**’s descriptions or instructions*”. These categories were merged post-hoc after discovering that the ambiguity of the two definitions led to poor inter-annotator agreement on the individual categories.

Model	Appropriate	Maybe		N/A
		Appropriate	Inappropriate	
seq2seq	87.0	11.0	0.0	2.0
+ global & local	87.0	12.0	0.0	1.0
human	97.0	2.0	0.0	1.0

Table 3.5: Percentage of utterances deemed appropriate by majority vote across 3 human evaluators.

even the improved model failed to capture this, mainly generating instructions even if it was inappropriate or unhelpful to do so. We explore how we can use these Architect dialogue acts to enrich our data in Chapter 4.

Appropriateness Evaluators were asked to rate the appropriateness of an utterance by categorizing the appropriateness of the *type* of utterance in the game context into one of the following categories:

- *Appropriate*: the type of utterance is a completely reasonable response given the preceding dialogue; e.g., if a question was asked, the utterance answers it; if confirmation is requested, the utterance provides it; etc.
- *Maybe appropriate*: the type of utterance could be considered a reasonable response given the preceding dialogue; though it may not be the most natural or polite option, it is not clearly an incorrect type of response that should be elicited from the dialogue.
- *Inappropriate*: the type of utterance is clearly incorrect given the preceding dialogue.
- *N/A*: the utterance cannot be evaluated for appropriateness (due to disfluency).

Appropriateness results are shown in Table 3.5 ($\alpha = 0.588$). Because of the tendency for models to routinely generate instructions, model responses were seen as slightly inappropriate and dismissive of the dialogue context. On the other hand, human responses, containing a wider spread of dialogue act types, were almost universally seen to be appropriate in context.

Executability Evaluators were asked to rate the executability of instruction-type utterances in the current game state. This criterion aimed to analyze the feasibility of instructions generated by models, regardless of whether the instruction led the Builder towards task success. For instruction-type utterances, evaluators were asked to select one of the following categories:

Model	Perfectly Clear	Somewhat Unclear	Unclear/ Impossible
seq2seq	63.6	29.9	6.5
+ global & local	61.6	30.1	8.2
human	91.7	8.3	0.0

Table 3.6: Percentage of instruction-type utterances deemed executable by majority vote across 3 human evaluators. Instruction-type utterances are identified by majority vote of annotated dialogue acts.

- *Perfectly clear*: given the current state of the board, the instruction is clear enough such that it can be immediately executed by **B**; i.e., all references to blocks, shapes, colors, spatial relations, etc. in the utterance constitute a description that is consistent with and executable in the current game.
- *Somewhat unclear*: the blocks/features described in the instruction are consistent with the current game state, but the instruction itself is ambiguous or underspecified and is therefore not immediately executable.
- *Completely unclear or impossible*: the instruction describes blocks/features that are not consistent with the current game state, or is impossible to execute in the current game state.

Executability results are shown in Table 3.6 ($\alpha = 0.860$). While humans here have a lower rate of generating instructions (see Table 3.4), the instructions they do produce are almost always perfectly executable.

Utterance correctness In addition to the colors, spatial relations, and other entity properties mentioned in the utterance, evaluators were asked to rate the correctness of the utterance with respect to the target structure and overall task goal. Here, it is important to note that an utterance can be fully correct without necessarily needing to be immediately executable: e.g., “*We’re going to build a row of 3 green blocks*” may not be specific enough for **B** to immediately take action (*maybe executable*), but it can be correct with respect to the target structure if said structure contained such a row that could feasibly be placed at that point in the game (*fully correct*). Additionally, placement of temporary blocks may be necessary to eventually build “floating” (suspended) blocks; these could also be deemed correct invariant of color as long as their relative placement followed a reasonably efficient path towards the target.

Model	Fully Correct	Partially Correct	Completely Incorrect	N/A
seq2seq	14.0	28.0	48.0	10.0
+ global & local	25.0	36.0	32.0	7.0
human	89.0	2.0	0.0	9.0

Table 3.7: Percentage of utterances deemed correct by human evaluators.

Given a window of game context (consisting of at least the last seven Builder’s and Architect’s actions, but always including the previous Architect’s utterance) and access to the target structure to be built, evaluators were asked to rate the correctness of an utterance immediately following that context with respect to task completion according to the following criteria:

- *Fully correct*: all elements of the instruction that are described (colors, spatial relations) that should be a part of the final structure are consistent with the target, and all references the instruction makes to existing blocks or structures are consistent with the current state of the world.
- *Partially correct*: some elements of the instruction that are described (minimally, the type of action and the color of the block to be used) are correct with respect to the overall target structure, while other elements are incorrect. With some minor corrections to the utterance, the instruction can be seen as close to being fully correct with respect to the target, but slightly misses the mark.
- *Completely incorrect*: the elements described in the instruction are completely incorrect with respect to the target structure.
- *N/A*:³ the utterance does not contain enough information to be judged for correctness.

Results can be found in Table 3.7. Unsurprisingly, without access to world state information, the baseline model performs poorly, conveying incorrect information about half of the time. With access to a simple world representation, our full model shows marked improvement on generating both *fully* and *partially* correct utterances. Finally, human performance sets a high bar; when not engaging in chitchat or correcting typos, humans consistently produce fully correct utterances constructive towards task completion.

³Originally, *N/A* was used to indicate either that an utterance was not informative enough or that it had already been disqualified due to being an unclear/impossible instruction (see previous section on Executability). Those utterances deemed non-executable and also marked as *N/A* were modified to be labeled as *Incorrect* in a postprocessing step.

3.6.3 Qualitative Analysis

Here, we use examples to illustrate different aspects of our best model’s utterances.

Identifying the game state In the course of a game, players progress through different states. In the human-human data, dialogue is peppered with context cues (greetings, questions, apologies, instructions to move or place blocks) that indicate the flow of a game. Our model is able to capture some of these aspects. It often begins games with an instruction like *“we’ll start with blue”*, and may end them with *“ok we’re done!”* (although it occasionally continues with further instructions, e.g. *“great! now we’ll do the same thing on the other side”*.) It often says *“perfect!”* immediately followed by a new instruction which indicates the model’s ability to acknowledge a Builder’s previous actions before continuing. The model often describes the type of the next required action correctly (even if it makes mistakes in the specifics of that action): it generated *“remove the bottom row”* when the ground truth was *“okay so now get rid of the inner most layer of purple in the square”*.

Predicting block colors and spatial relations Generated utterances often identify the correct color of blocks, e.g. *“then place a red block on top of that”* in a context when the next placements include a layer of red blocks (ground truth utterance: *“the second level of the structure consists wholly of red blocks. start by putting a red block on each orange block”*.) Less frequently, the model is also able to predict accurate spatial relations (*“perfect! now place a red block to the left of that”*) for referent blocks.

Utterance diversity and repetition Generated utterances lack diversity: the pattern *“a x b”* (for a rectangle of size $a \times b$) is almost exclusively used to describe squares (an extremely common shape in our data), often appearing in the form *“we’re going to build a 3x3 square”* and almost always incorrectly and inappropriately. Utterances are mostly fluent, but sometimes contain repeats: *“okay, on top of the blue block, put a blue block on top of the blue”* or *“yes, now, purple, purple, purple, ...”* Notably, the most fluent utterances generated by the model typically span single-block instructions and follow very generic, safe templates such as *“now/then place a [color] block on top of that”*, *“do the same thing on the other side”*, *“and one to the right of that”*, or *“sorry, it should be on the other side”*, despite the model being trained on highly varied human **A** utterances that may span multiple blocks (*“place a row of 5 orange”*, *“now put a tower of 3 orange blocks on the left side then delete the bottom 2”*) and contain more descriptive ways of referring to the world (*“and just move it to the left of the highest purple block”*, *“now 1 more going diagonally up and to the left from the 1 you just placed”*).

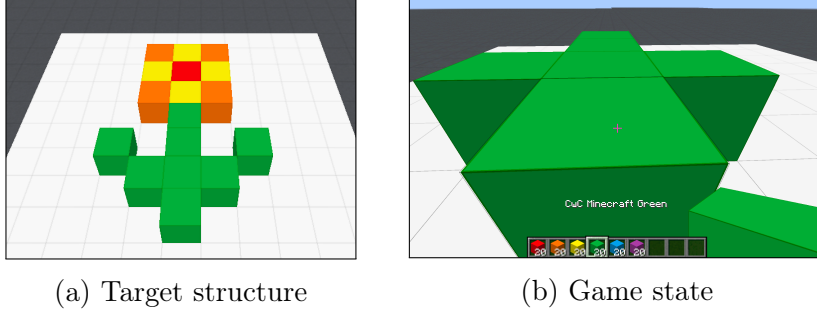


Figure 3.6: Example 3.1: target structure vs. current world state. The utterance generated here is *“and then put a yellow block on top of that”*, which in this context is partially correct (yellow blocks do need to be placed next, but not in the location described).

Here, we provide a couple of examples of utterances generated by our model, placed within the context of the game state with accompanying screenshots.

Example 3.1. This example shows a game in which the target in Figure 3.6a is being built. At this point in the game (Figure 3.6b), the model generates *“and then put a yellow block on top of that”*. While the generated spatial relation is incorrect with respect to the target structure, the color of the mentioned block (yellow) is correct.

Example 3.2. This example shows a game in which the target in Figure 3.7a is being built. At this point in the game (Figure 3.7b), the Builder has not placed any blocks yet. In this instance, the model generates *“place a red block on the ground”*. In this case, both the color of the mentioned block (red) and its spatial relation with respect to the ground are correct.

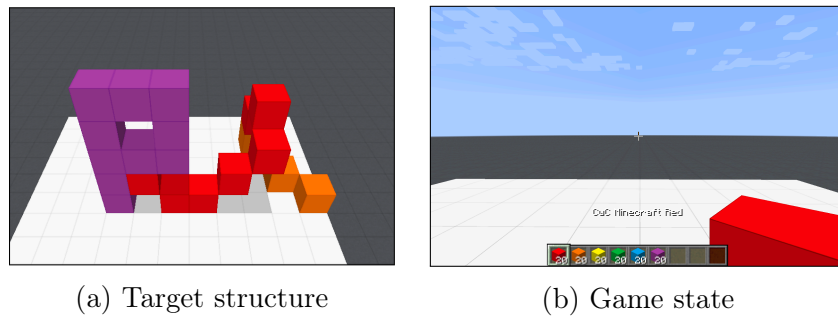


Figure 3.7: Example 3.2: target structure vs. current world state. The utterance generated here is *“place a red block on the ground”*, which is fully correct.

3.7 BLOCK COUNTERS V2

In the previous sections, we described and evaluated the original Architect block counters model as published in ACL 2019 [39]. Motivated by the need to further improve the quality and robustness of the model’s outputs in order to deploy it in interactive demos (described in more detail in Chapter 6), we have since made additional improvements to model performance by fixing bugs in the original codebase, filtering the training data of ungrammatical utterances, and applying domain-specific constraints during decoding. In this section, we describe these changes and their impact in detail.

Bug fixes We discovered and fixed a number of bugs in our original modeling codebase. Specifically, during beam search decoding, end-of-sentence token scores were not being taken into account properly when scoring complete sentences, leading to strange behavior with fragmented outputs. Additionally, we addressed issues involving experiment reproducibility, which was previously variable due to improper seeding of random number generators and spawning of training subprocesses.

Data cleaning When analyzing our model’s outputs qualitatively, we noticed a significant number of instances where the model would get stuck in a repetitive loop, generating comma-separated lists of colors in a nonsensical manner (e.g. “*okay, now, purple, empty, purple, purple, ...*”). These appeared similar to instances in the training data involving one participant, Architect ID 44 (A44), who frequently employed a nonstandard approach to describing instructions for planar target structures: first describing the structure as a “*2d drawing*,” they proceeded to list block colors in a column-by-column fashion, e.g.:

A: ok another 2d vertical drawing

...

A: empty, 2 red, 2 green. start anywhere moderately left

...

A: empty 2 red, 2 empty, 1 green

...

A: 2 empty on the not 1 empty. before the green

...

A: 2 spaces between the red and green on the second column

...

A: wait stop I’m making a correction

B: *yeah I don't follow*
 B: *can u rephrase?*
 A: *on the second column. I said empty, 2 red, 2 empty, green*
 ...
 A: *3rd column: 3 empty, 3 green*
 ...
 A: *2red, 1 green, 3 empty, 1 green*
 ...
 A: *2 red*
 ...
 A: *we're done. looks like a cherry goodj ob*
 B: *ahh*

In these dialogues, this Architect stylistically deviated from the norm by using formulaic, repetitive, list-like instructions instead of describing intuitive patterns or concepts. While ideally we would like our model to be robust to these kinds of situations (as human Builders are still able to interpret them), we saw that including these utterances during training negatively impacted the model’s ability to produce fluent output. Overall, we identified 13 dialogues involving this participant that exhibited these properties. We therefore omit the 7 training set dialogues in which these instructions occur (but keep the 6 development and test set dialogues for evaluation). After cleaning, our training split contains 6,422 samples (previously 6,548 samples).

Constrained decoding Even though the block counters model showed improved correctness when generating utterances (and, particularly, on its usage of color terms), model outputs still sometimes mentioned colors that were clearly incorrect given knowledge of the current built and target structures (e.g., instructing to place a *green* block when *Target* contained no green blocks). To combat this, we leverage knowledge of the set of colors that are actually valid at a given point in a game to filter candidate utterances as they are produced by the model. We implement this by discarding candidate hypotheses generated during the beam search process subject to the following criteria:

- **Color existence:** Given *Built* and *Target*, we know the sets of colors that exist in both. If a candidate hypothesis includes a color that does not exist in either of these sets, it is discarded. For example, a candidate is filtered if it mentions the color *blue* but there are no blue blocks in either *Built* or *Target*.

- **Referent block existence:** We discard a candidate hypothesis if it mentions a referent block with a color that does not exist in *Built*. For instance, a candidate is filtered if it refers to a location *on top of the yellow block* but there are no yellow blocks currently in *Built*. For every invalid color *c*, we identify illegal referent block colors in candidate hypotheses using the following regular expressions:

```
(on top of|in front of|behind) (article)(referent_modifier)? c
(on|to the)? (left|right)( side)? of (article)(referent_modifier)? c
```

where

```
article          = (the|these|those|that|this)
referent_modifier = ( last| top| topmost| left| leftmost| right|
                    rightmost| bottom| bottommost| middle)
```

- **Feasible placements:** The global feasible next placements block counter (as defined in Section 3.4) indicates counts of colors within the entire build region that can feasibly be placed next. We discard a candidate hypothesis if it contains an instruction to place a block with an infeasible color (that is, the next placements counter for that color is 0). For example, a candidate is filtered if it says to *place a purple block*, but the feasible next placements counter indicates that, across all optimal alignments of *Built* and *Target*, no purple blocks can be feasibly placed next. For every invalid color *c*, we identify illegal instructions in candidate hypotheses using the following regular expression:

```
(put|place|add)( a| an| another| one)? c
```

Increasing the local counter region size The local block counters in our original model only consider the 27 cells in the $3 \times 3 \times 3$ region concentrated around the Builder’s last action, each represented as a set of 18-dimensional block counters, when encoding spatial information about this region. We also experiment with expanding this region size to include more locations outside the immediate neighborhood of the Builder’s last action. In particular, we increase the region size to include the $7 \times 7 \times 7$ cube around the Builder’s last action, where each of the immediately adjacent cells, adjacent rows/columns, quadrants, and octants are represented as its own set of 18-dim block counters. The block counters for the resulting 33 regions are again concatenated in a deterministic fashion with respect to the Builder’s position and gaze. However, we found that training models with local counters of increased size on the original dataset resulted in slightly worse performance across almost all automated metrics.

Model	BLEU				Precision / Recall				spatial	dialogue
	B-1	B-2	B-3	B-4	all keywords		colors			
Baseline	16.2	8.1	4.7	2.8	14.5 / 13.8	14.8 / 23.3	10.7 / 9.5	17.9 / 20.6		
+ bug fixes	17.6	8.6	5.0	3.0	16.0 / 14.9	17.5 / 22.9	13.1 / 9.8	18.8 / 22.3		
+ clean data	18.0	9.2	5.3	3.3	15.7 / 15.3	21.2 / 28.5	11.1 / 9.8	19.0 / 22.3		
+ constraints	17.7	8.9	5.1	3.1	15.5 / 15.0	27.2 / 28.9	11.8 / 9.4	18.0 / 21.4		

Table 3.8: Updated BLEU and term-specific precision and recall scores for the ablation study on the development set.

Model	BLEU				Precision / Recall				spatial	dialogue
	B-1	B-2	B-3	B-4	all keywords		colors			
baseline	15.7	8.1	4.8	2.9	13.5 /	14.4	14.9 /	28.7	8.7 /	8.7
counters v2	17.4	9.2	5.6	3.6	15.0 /	15.0	25.9 /	30.2	10.8 /	8.4
									16.6 /	21.0

Table 3.9: Updated BLEU and term-specific precision and recall scores on the test set.

3.7.1 Experimental Setup

As before, we use 300-dimensional Gated Recurrent Units (GRUs) [48] for all RNN modules and use 300-dimensional pretrained GloVe word embeddings [46]. All linear layers were initialized using Xavier initialization [64]. All RNNs have a hidden state size of 300. In cases where we use a bidirectional encoder RNN, the sum of the two final hidden states in either direction constitutes the final encoding of dialogue history (used to initialize the decoder). See Appendix D.2 for detailed hyperparameter configurations.

Because the bug fixes and data cleaning significantly affected the way these models were trained and evaluated, we re-ran a grid search over hyperparameters for each of these settings in our ablation study. During the grid search, we explored variants of block counters model architectures in which the hyperparameters for RNN modules were varied and the global and local block counters were combined in slightly different ways. As a result, the baseline model architecture originally reported in Section 3.4 is somewhat obsolete; we note the architectural differences between the baseline model and subsequent iterations in Appendix D.2.

We use constrained decoding as a final postprocessing step that is applied to only our best model. We therefore only show the effect of applying constrained decoding to the best model that was trained with bug fixes and cleaned data.

3.7.2 Results and Analysis

We analyze the effect of each of the above features by performing an ablation study on the development set using the numbers originally reported in Section 3.6 as our baseline.⁴ We also show the difference in test set performance between the baseline and fully updated and improved model (block counters v2).

Ablation study Table 3.8 shows the effect of applying bug fixes, data cleaning, and constrained decoding to the baseline model. The decoding bug fixes net a modest gain in BLEU scores. Using cleaned data during training widens the gap, while also contributing a significant boost to color precision and recall (since the model is now less likely to produce long lists of erroneous colors). Finally, constrained decoding results in a large gain in color precision over all previous models at the cost of a slight hit to BLEU scores and other terms. Because generating colors that are consistent with the built and target structures is so important to task success, the gains here significantly outweigh the costs.

Test set results We compare the baseline model (the final, finetuned block counters model originally reported in ACL 2019) to the block counters v2 model on the test set in Table 3.9. Our updated model shows modest improvements across most terms, with significant improvements on color terms.

3.7.3 Qualitative Analysis

While we were not able to perform a full human evaluation of these updated block counter models, we perform here a brief qualitative analysis of that model’s output utterances and compare them to previous results.

Nonsensical utterances As a direct result of data cleaning, the updated model no longer produces nonsensical, repetitive utterances. In one example, our original block counters model generated “*on top of each purple , purple , empty , purple , empty , purple , empty , purple , empty , purple , empty , purple , empty , purple , empty , purple , empty ,*”, while our updated model simply produces “*put a purple block on top of that one*” instead.

⁴Because of our reproducibility bug fixes, the baseline numbers are not directly reproducible without reverting a large amount of code. We therefore report the original numbers for the sake of comparison.

Filtering incorrect colors Constrained decoding directly prevents our improved model from producing utterances that are clearly incorrect. In one example, despite the feasible block counters indicating that only orange blocks can be placed next, our original model still generates “*place one green on top of that*”; meanwhile, our updated model is not allowed to produce such an utterance, and instead generates “*place one orange on top of that*”. In this case, the human **A** had instructed **B** to “*now place one orange on top of the orange on each end*”, which more closely aligns with our updated model’s output.

In another example, our original model produces “*then place a blue block on top of that*” despite the global block counters indicating only 3 orange blocks remain to be placed to finish building *Target*. On the other hand, our updated model produces an utterance that mentions the correct color, *orange*, but is more convoluted: “*great , now we ’re going to make a 3 x 3 orange square on the ground in the middle of the square*”. In this case, the actual human **A** utterance is “*great now just three orange blocks left.*” While the colors mentioned by our updated model are at least correct with respect to the remaining blocks in *Target*, the rest of the utterance remains largely incorrect. This shows that while constrained decoding is a convenient fix for easy-to-identify color errors, it does not address deeper issues of utterance correctness, which necessitates a move away from representing the world using simple block counters altogether.

3.8 USING ORACLE NEXT ACTIONS

Until now, our block counters models have assumed that the Architect does not need to explicitly plan the next action or series of actions to describe to the Builder next. This requires our model to leverage the world state representation to not only decide how to describe what needs to be done next using natural language, but also determine what the optimal next sequence of actions to transform *Built* to *Target* should be. Additionally, in their current implementation, we force the local block counters to attend to and summarize the localized region surrounding **B**’s last action. However, this world state information may not be sufficient if the true next action to describe is not clearly defined within that region.

We thus wish to investigate how our utterance generation model might improve if given an idea of where the next **B** action should take place. This can be likened to adding an explicit action planning component to the model. Since the Architect always has access to *Target*, a naive heuristic could be to simply use a random feasible action based on the current *Built* state, or to choose the feasible action that is closest to **B**’s last action. Ideally, the action planning step could be learned, such that an external module predicts where the next action should occur. By taking this approach, we can decouple the problem-solving and utterance

generation subtasks, allowing our model to better focus on the singular task of generating natural language utterances given a specific action plan and offloading the burden of building that plan to an external module.

To this end, we investigate model performance when we instead provide additional information about the immediate next **B** action to the block counters model in an oracle experiment. For a given point in the game at which the human **A** spoke next, we compute the oracle next **B** action by looking forward through the game log until **B**'s next block placement or removal and provide this as additional input to our block counters model. In this section, we describe how we modify the block counters v2 model to instead use **B**'s next action and examine its effect on utterance generation performance.

Representing oracle next actions In order to re-use the block counters architecture from Section 3.4, in addition to computing the standard global block counters, we compute local block counters with respect to the oracle next action instead of **B**'s last action. That is, we consider the $3 \times 3 \times 3$ cube of block locations directly surrounding the oracle next action, as well as the action itself, and compute a separate set of block counters for each of these 27 locations.

For each grid cell, in addition to the original 18-dimensional block counters, we also explicitly encode information about its relative position with respect to both the last **B** action and **B**'s current position and orientation. Given a cell c with absolute coordinates $\langle x_c, y_c, z_c \rangle$ and the last **B** action with absolute coordinates $\langle x_l, y_l, z_l \rangle$, we compute the difference between the two locations, $\langle x_d, y_d, z_d \rangle = \langle x_c - x_l, y_c - y_l, z_c - z_l \rangle$. Following Jayannavar et al. (2020) [38], we also calculate the relative perspective coordinates $\langle x'_c, y'_c, z'_c \rangle$ with respect to **B**'s current position $\langle x_B, y_B, z_B \rangle$ and orientation (pitch $\phi_B \in [-90, \dots, +90]$, or vertical rotation, and yaw $\gamma_B \in [-180, \dots, +180]$, horizontal orientation) by moving the frame of reference from $\langle 0, 0, 0 \rangle$ to $\langle x_B, y_B, z_B \rangle$, and rotating it to account for **B**'s yaw and pitch:⁵

$$\langle x'_c, y'_c, z'_c \rangle = P \cdot Y \cdot \langle x_c - x_B, y_c - y_B, z_c - z_B \rangle \quad (3.11)$$

This results in an additional 6-dimensional vector of coordinates per cell. We scale both sets of coordinates by a factor of .1 to keep their range closer to that of the block counters and concatenate them to the block counter representation, resulting in a 24-dimensional representation per grid cell. As before, relative directions for each location are assigned based

⁵ $P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \phi_B & \sin \phi_B \\ 0 & -\sin \phi_B & \cos \phi_B \end{pmatrix}$ and $Y = \begin{pmatrix} \cos \gamma_B & 0 & -\sin \gamma_B \\ 0 & 1 & 0 \\ \sin \gamma_B & 0 & \cos \gamma_B \end{pmatrix}$

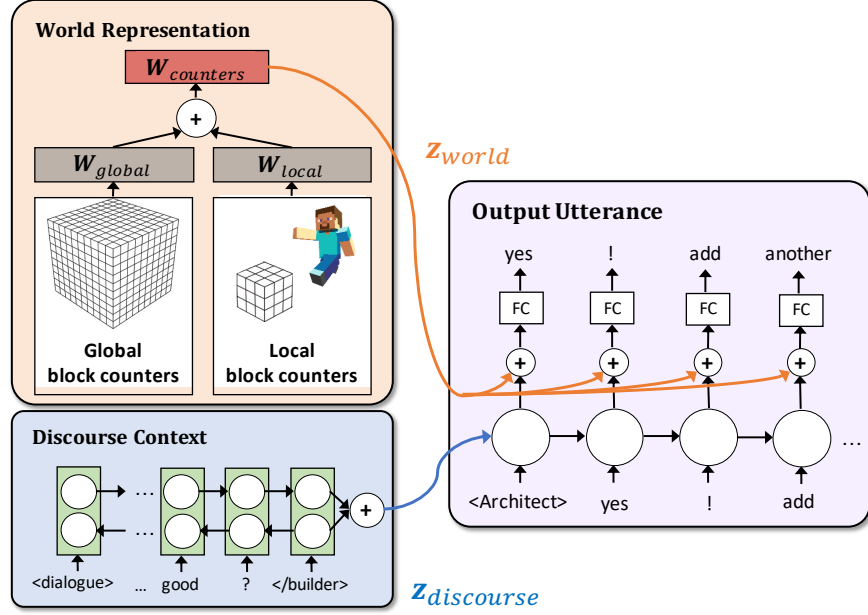


Figure 3.8: The modified block counters v2 model architecture used with oracle next **B** actions. In this case, the local block counters are computed using the next **B** action as a reference (instead of the last **B** action).

on **B**’s position and gaze and used to concatenate the 27 24-dimensional cell representations in a fixed canonical ordering.

Adding block counters to the model We use a slightly different strategy than before to add block counters to our models. Instead of adding the embedded block counters to the decoder’s input, we found the best results by adding the block counters embedding to the decoder’s output at every time step and feeding the result through a network of fully connected layers, the output of which is then fed to a final linear layer to generate the next token (see Figure 3.8).

3.8.1 Results and Discussion

For our oracle experiment, we compare our updated model that uses oracle next actions to the block counters v2 model described in Section 3.7. Both models use 1-layer bidirectional encoder RNNs for the dialogue history, train with a dropout of 0.5, and use constrained decoding during generation. See Appendix D.3 for detailed hyperparameter configurations.

Tables 3.10 and 3.11 show the result of adding oracle next **B** actions to the block counters v2 model on the development and test sets, respectively. By adding next action information, the model enjoys modest gains in BLEU score and term precision and recall over all keywords

Model	BLEU				Precision / Recall					
	B-1	B-2	B-3	B-4	all keywords	colors	spatial	dialogue		
counters v2	17.7	8.9	5.1	3.1	15.5 / 15.0	27.2 / 28.9	11.8 / 9.4	18.0 /	21.4	
next actions	19.5	9.6	5.7	3.5	17.9 / 16.9	28.0 / 34.9	12.6 / 11.2	21.8 /	19.2	

Table 3.10: BLEU and term-specific precision and recall scores on the development set using oracle next **B** actions.

Model	BLEU				Precision / Recall					
	B-1	B-2	B-3	B-4	all keywords	colors	spatial	dialogue		
counters v2	17.4	9.2	5.6	3.6	15.0 / 15.0	25.9 / 30.2	10.8 / 8.4	16.6 /	21.0	
next actions	18.6	9.3	5.5	3.5	16.8 / 17.6	23.8 / 36.0	11.8 / 11.0	19.8 /	18.8	

Table 3.11: BLEU and term-specific precision and recall scores on the test set using oracle next **B** actions.

and spatial terms. The model also enjoys a sizable boost to color recall on both development and test sets. In Section 4.6, we will return to a block counters model that uses oracle next **B** actions and analyze its performance in greater detail.

The promising initial results from focusing our world state representation around the next action speaks to the need for having an explicit action predictor component for the Architect. To move away from oracle experiments and investigate the feasibility of adding next action information to a fully interactive model, a logical next step is to instead train a next action predictor model to explicitly plan **A**’s next actions. At the time of this writing, we do not have such a model; future work could explore how to adapt an existing action prediction model (such as the BAP model of Jayannavar et al. (2020) [38]) to an Architect action prediction task, which has access to both *Built* and *Target*.

3.9 CONCLUSION

In this chapter, as a first step towards building fully interactive Architect models, we defined the Architect utterance generation task and described our initial models for this task. At its core, our model used a sequence-to-sequence architecture, processing the dialogue history as a flat input sequence of tokens and decoding the next Architect utterance in a token-by-token fashion. Our model also included a simple representation of the world state, encoded as *block counters*, which compared the built and target structures in order to

summarize information about the colors of blocks to be placed and removed.

In addition to the standard BLEU metric, we proposed our own keyword-based automated metrics, *term-specific precision and recall*, to estimate measures of task success, and showed that our models that utilized the block counter representation performed markedly better on these terms (and in particular, colors) than naive baselines. We also proposed a thorough set of criteria for human evaluation to judge both utterance-level and task-specific aspects of generated utterances, and showed that our block counters models showed promising improvement on utterance correctness. By addressing some of the obvious issues that arose from qualitative analysis of these models, we then trained an updated version of these models with more robust results, and saw further improvement by incorporating oracle information about **B**'s next actions to this updated model.

However, human evaluation and qualitative analysis both show that there is still a large gap between our model's and human performance. This can perhaps be attributed to the fact that our models must learn how to use different kinds of utterances appropriately, refer to higher-level concepts in the world, generate spatial relations grounded to the Builder's perspective, compare built and target structures to determine next steps, and other skills from a limited amount of noisy human-human data in an end-to-end fashion. In the next chapters, we attempt to address some of these issues by adding enriched linguistic structure in the form of dialogue acts (Chapter 4) and building better world representations by locating shapes in the Minecraft world (Chapter 5).

Chapter 4: Adding Linguistic Structure: Dialogue Acts

In the previous chapter, we introduced a series of baseline neural models for generating Architect utterances within a given context. While models that included richer representations of the world showed improved correctness of generated utterances over naive baselines, all models exhibited problems with generating diverse dialogue acts: model outputs often tended towards generic, single-step instructions, ignoring the need to respond to questions or provide corrections (see Table 3.4 in Chapter 3). On the other hand, human responses in our data are much more diverse: humans give noticeably fewer instructions, instead providing many more appropriate corrections, verifications, and answers to questions in order to guide the Builder more effectively toward the goal. Capturing these elements of the Architect’s job is key to a successful dialogue interaction, as we will see in Chapter 6.

When evaluating our models using human judgments, we showed that Architect dialogue acts can be limited to a few coarse-grained categories and annotated with a reasonable level of inter-annotator agreement (Section 3.6.2). Motivated by these promising annotation results, we propose here to explicitly label and model **Architect dialogue acts** in our data. Because Architects can include a variety of information in their utterances, and utterance boundaries are not limited to single sentences, annotating Architect dialogue acts is a multi-label problem in which a given utterance can be classified as any number of dialogue acts from a candidate set. This is in contrast to Builder utterances, which are shorter, simpler, and usually incorporate only a single Builder dialogue act such as asking for clarification or verification or acknowledging the Architect’s instructions. While we have also annotated Builder utterances in our corpus with dialogue acts and explored the task of classifying them, we focus only on Architect dialogue acts in this thesis, and leave a more thorough exploration of Builder dialogue acts to future work.

In this chapter, we use the dialogue acts previously defined for human evaluation of our utterance generation models as a launching point for annotating our entire corpus with Architect dialogue acts, which we slightly redefine in Section 4.2. Then, we analyze the difficulty of both classifying past dialogue acts (Section 4.3) as well as predicting the next dialogue act (Section 4.4) in a given context by showing baseline results for these isolated tasks. Finally, we explore using these dialogue acts to condition a model that generates Architect utterances in Section 4.5, and add oracle next Builder actions to such a model in Section 4.6.

Speaker	Dialogue Act	Utterance
A	YES-NO-QUESTION	So do you go to college right now?
A	ABANDONED	Are yo-,
B	YES-ANSWER	<i>Yeah,</i>
B	STATEMENT	<i>it's my last year [laughter].</i>
A	DECLARATIVE-QUESTION	You're a, so you're a senior now.
B	YES-ANSWER	<i>Yeah,</i>
B	STATEMENT	<i>I'm working on my projects trying to graduate [laughter].</i>
A	APPRECIATION	Oh, good for you.
B	BACKCHANNEL	<i>Yeah.</i>
A	APPRECIATION	That's great.
A	YES-NO-QUESTION	um, is, is N C University is that, uh, State,
B	STATEMENT	<i>N C State.</i>
A	SIGNAL-NON-UNDERSTANDING	What did you say?
B	STATEMENT	<i>N C State.</i>

Figure 4.1: A fragment of labeled conversation in the Switchboard corpus, labeled according to the SWBD-DAMSL tagset [71].

4.1 BACKGROUND

The identification and use of dialogue acts to model discourse structure has long been studied in linguistics and NLP. Dialogue acts follow from the insight that an utterance in a dialogue is a kind of action performed by a speaker [66, 67, 68, 69]. In his theory of speech acts, Austin (1962) [67] categorizes utterances in speech situations under three types of acts; in particular, dialogue acts are those that involve illocutionary force: i.e., the act of asking, answering, promising, etc. in uttering a sentence. Furthermore, Searle (1975) [70] builds a taxonomy of speech acts focused on illocutionary forces, categorizing them into the five major classes of assertives, directives, commissives, expressives, and declarations.

Dialogue acts [72] (or, alternatively, dialogue moves or conversational moves [73, 74]) are an extension of speech acts that additionally model other conversational features, such as grounding the speaker and hearer to a common ground. Thus, dialogue acts can be thought of as a tagset used to classify utterances based on pragmatic, semantic, and syntactic criteria. Some effort has been made to develop domain-independent dialogue act labeling systems, such as the Dialogue Act Markup in Several Layers (DAMSL) architecture [75], which tags utterances for forward-looking functions (such as speech acts) and backward-looking functions that look back to the previous utterance in the dialogue (such as grounding and question answering). However, proposed dialogue act tagsets are generally domain-dependent. For example, the SWBD-DAMSL tag set [71] is a multidimensional extension of DAMSL specifically designed for annotating utterances in the Switchboard corpus of spontaneous human-human telephone speech [76] (Figure 4.1). Another example includes the dialogue act taxonomy designed for the VERBMOBIL system [77], a prototype for translating spoken dialogues between users who want to agree on a date for a business meeting (Figure 4.2).

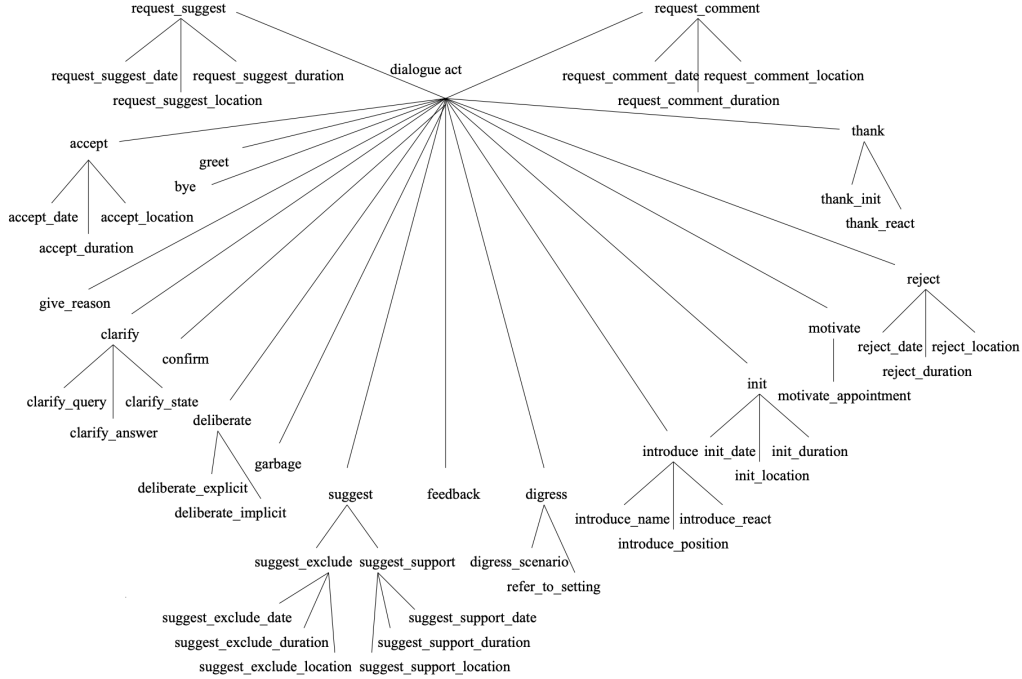


Figure 4.2: Taxonomy of dialogue acts in VERBMOBIL [77].

Closely related to the dialogue acts we define for Minecraft dialogues, Carletta et al. (1997) define conversational moves in the HCRC Map Coding Scheme [74] for use on the Map Task Corpus [29] (Figure 4.3). Conversational moves are defined as different kinds of initiations, which set up discourse expectations, and responses, which fill those expectations, and are classified according to their conversational purposes. At a high level, the HCRC Map Coding Scheme includes conversational moves that could be broadly defined as instructions, clarifications, confirmations, questions and answers. However, these moves are more fine-grained, distinguishing between yes/no questions and other types, as well as different methods of clarifications, including those classified as initiations (e.g. explanations and checks) vs. responses (e.g. responses to clarification questions).

One way to interpret dialogue acts is to model them in a supervised classification task [69], in which dialogue act classifiers are trained on a corpus hand-labeled with dialogue acts. The converse task of generating dialogue acts is more difficult, but can be done in a rule-based manner, such as the update rules of Stent (2002) [78] for dialogue act generation in the TRIPS system [79]. More recent dialogue systems instead use a more advanced model for dialogue management called the information-state architecture [80, 81], which consists of several components: the information state (or discourse context), a dialogue act interpreter, a dialogue act generator, a set of update rules to update the information state and produce

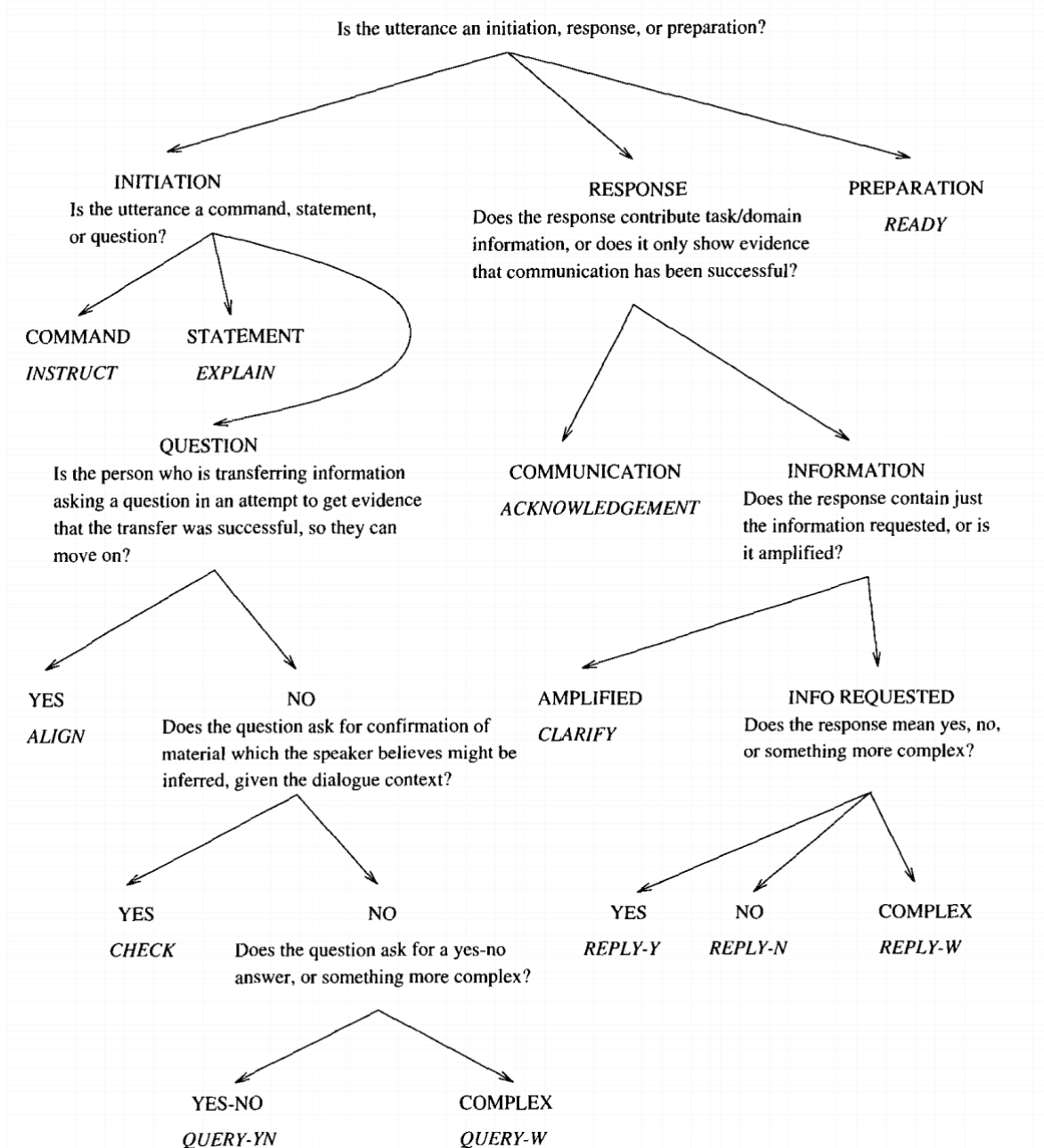


Figure 4.3: Summary of conversational moves in the HCRC Map Coding Scheme [74].

relevant dialogue acts, and a control structure to select which update rules to apply. These types of models are often learned probabilistically using Markov decision processes (MDPs) or partially observable Markov decision processes (POMDPs) [69, 82].

Since our approaches to modeling dialogue acts are much simpler, we do not discuss in detail the technical background behind information-state tracking here. In this chapter, we show that using simple classifiers to model dialogue acts that are much more coarsely defined than those of previous systems is still extremely beneficial for our Architect systems.

4.2 ADDING DIALOGUE ACTS TO THE MINECRAFT DIALOGUE CORPUS

In the Collaborative Building Task, the Architect’s role is multifaceted: while their main job is to provide clear, grounded instructions to the Builder, they must also be able to describe the target structure in meaningful terms, verify the built structure against the target, identify and rectify mistakes, and respond to the Builder’s questions and requests for clarification. Previously, in Narayan-Chen et al. (2019) [39], we evaluated this aspect of our data on a small scale by defining a set of 6 coarse-grained Architect dialogue acts and asking human evaluators to label both human- and model-generated utterances with these categories. To annotate our entire corpus, we define the set of Architect dialogue acts, with slight modifications to their original definitions, as follows:

- *Instruct B (*I*)*: **A** instructs **B** to move, place or remove blocks, or otherwise execute some action within the game using their player character (*“ok place a red block where you’re standing”*). These utterances can be direct commands (*“Can you punch out the two blocks on the bottom left?”*) or, less commonly, declarative sentences that imply that **B** should do something (*“now there is another “stair step” on top of that”*).
- *Describe Target (*D*)*: **A** describes the target structure or substructures within it (*“I think it’s a robot chicken. so now the wings.”*). At the beginning of a game, descriptions can also be somewhat abstract (*“so this looks like someone tried to spell with blocks and failed”*). Descriptions often co-occur with instructions, as **A** uses concepts or substructures in *Target* to describe the next executable step (e.g., *“basically like the orange ones”* and *“so it looks like a cross”* are labelled as both instructions and descriptions).
- *Answer question (*A*)*: **A** responds to a **B** question (**B**: *“here?”* **A**: *“yeah”*). Since **B** often asks **A** to check whether they’ve executed an instruction correctly, answers

to questions commonly co-occur with verifications (when execution is correct) and clarifications/corrections (when execution is incorrect).

- *Verification (*V*)* (renamed from *Confirm B’s actions or plans*): **A** verifies the built structure against the target and provides an explicit confirmation of the actions **B** has taken (“*perfect!*”, “*great*”). This category was changed from its previous definition to only include utterances where **A** verbalizes that what **B** has built or plans to build is *correct* when verifying it against *Target*. But if **A** notes problems with the structure after verifying *Built* against *Target* (“*no, sorry*”), it is instead labeled as a correction (previously labeled as a confirmation).
- *Clarification/Correction (*C*)*: **A** provides further clarification for ambiguous instructions or identifies and rectifies mistakes made by either player (“*sorry, i meant on top of the edge of the table*”). This category is an umbrella term for **A**’s clarifications and corrections with respect to their own utterances (**A**: “*Like that, except the side is one longer.*” **A**: “*Sorry, shorter*” **A**: “*not longer*” **A**: “*But perpendicular*” is a series of corrections, each elaborating on the last) as well as to **B**’s actions (“*move them up one space*” after a sequence of **B** actions). This also includes utterances where **A** simply indicates that a mistake has been made, but not how to resolve it (“*no, sorry*”), as well as follow-up utterances that simply elaborate on previous **A** utterances that were underspecified (**A**: “*Great. Now one block on the same level as the top blue block, on the side toward the purple structure*” **A**: “*Also blue*”). As a result, clarification/correction utterances can be highly varied.
- *Other (*O*)*: other utterance types not covered by the above categories, including but not limited to: offhand comments, chitchat, greetings, etc. (“*you too builder! you are a master craftsman/woman*”, “*alright on to the next one*”)

We annotate the entire set of 11,670 **A** utterances in the Minecraft Dialogue Corpus with these dialogue acts. On average, **A** utterances contain 1.29 dialogue acts with a standard deviation of 0.537. Table 3.4 shows the frequency of dialogue act types across the corpus, alongside the original percentages reported on a smaller test set reported in Section 3.6.2.

Annotation noise The original human evaluation assigned dialogue act labels to utterances based on majority vote between 3 evaluators, and as such could be evaluated for inter-annotator agreement. Unfortunately, our annotation of the corpus is not as clean; each Architect utterance was annotated by only one human annotator, resulting in many noisy annotations. While we can still use this data to build models that perform reasonably, the

Source (# samples)	Instruct B	Describe Target	Answer question	Verification	Clarification/ Correction	Other
Human eval. (100)	47.0	14.0	12.0	17.0	23.0	8.0
Full corpus (11,670)	58.0	11.8	12.3	21.9	18.5	6.4

Table 4.1: Percentage of utterances annotated as a given dialogue act in the Minecraft Dialogue Corpus. For completeness, we include again the percentage of 100 human utterances annotated as a given dialogue act in the previous human evaluation (Section 3.6.2) to show the similarity of the distributions of dialogue acts.

effect of annotation noise should not be ignored. We will return to this point repeatedly in the qualitative analyses of our models.

Single-label dialogue acts 74.8% of all **A** utterances consist of only a single label. Of these utterances, most are instructions (54.9%), then verifications (15.5%) and clarifications (10.9%). Even less common in isolation are chit-chat utterances (8.5%) and descriptions (7.2%). Lastly, answers to questions rarely appear alone (3.0% of single-label utterances), as they often appear alongside other dialogue acts (described in more detail below).

Multi-label dialogue acts Since **A** utterances in the Minecraft Dialogue Corpus can consist of multiple sentences addressing various needs, they can be categorized as any number of the above dialogue acts. For instance, in response to **B** completing a substructure and asking, “*like this?*”, **A** responds by simultaneously answering **B**’s question, verifying **B**’s actions, and providing new instructions while describing elements of the target structure:

*A: good job builder. if you can do this without a supporting structure please do. we need yet another row of purple blocks that is 9 long so that it lays across the green and red, on top of them *AVID**

More common are utterances representing a couple of closely-related dialogue acts, such as instructions and descriptions:

*A: so this purple is kind of like an x if that helps. we need to put purple blocks in the top right and lower right of that center purple block *DI**

To quantify the co-occurrences between dialogue acts, Table 4.2 depicts how often a given dialogue act appears either alone (*none*) or co-occurring with another type of dialogue act across all utterances of the given type. Instructions mostly appear in isolation, but can also be included as part of other utterances fairly equally. Descriptions, on the other hand,

Dialogue Act	none	Co-occurrence %					
		I	D	A	V	C	O
Instruct B	70.8	–	8.3	5.4	10.5	10.7	0.09
Describe Target	45.9	41.0	–	5.9	6.0	10.4	0.07
Answer question	18.2	25.6	5.6	–	38.7	37.1	0.07
Verification	52.9	27.8	3.2	21.8	–	3.6	0.0
Clarification/Correction	43.8	33.6	6.6	24.6	4.3	–	0.09
Other	99.2	0.80	0.13	0.13	0.0	0.27	–

Table 4.2: Percentage of total utterances of each dialogue act type that appear in isolation (*none* column) or co-occur with other dialogue act types (remaining columns). For each row in the table, the first column indicates how often an utterance of that type appeared in isolation; the following columns indicate how often an utterance of that type co-occurred with another given dialogue act.

primarily appear either alone or bundled with instructions. Answers to questions rarely appear as isolated dialogue acts, and co-occur with verifications and clarifications/corrections equally as often, underlining the need for **A** to understand game context in order to answer a question correctly. Verifications and clarifications/corrections behave similarly to each other, appearing both unprompted as well as in response to questions, and also often include further instructions. Finally, chit-chat utterances are almost always seen alone, suggesting that the human Architects in our corpus stay on-topic appropriately.

Simplified Architect acts Because descriptions of target structures appear frequently alongside regular instructions, the distinction between these two types of dialogue acts is sometimes fuzzy. Additionally, our current Architect utterance models do not yet capture well enough the higher-level concepts needed to generate adequate descriptions of target structures. To circumvent the problems that arise when modeling this particular dialogue act, we also experiment with a simplified set of 5 Architect dialogue acts in which the *Describe Target* category is merged into *Instruct B*. While we report dialogue act classification results on both sets in Section 4.3, we consider only the simplified set for models that perform next dialogue act prediction (Section 4.4) as well as utterance generation models that use those predicted dialogue acts (Section 4.5).

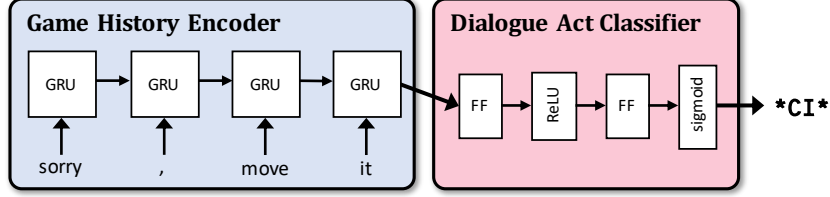


Figure 4.4: The Architect dialogue act classifier model.

4.3 DIALOGUE ACT CLASSIFICATION

Classifying Architect dialogue acts can be particularly useful for interactive Builder agents. For instance, a Builder agent that can identify Architect instructions in the dialogue history could use that information to determine when the agent should act, or how those instructions should align to corresponding sequences of Builder actions. Enriching the game history by incorporating a dialogue act classifier can also be useful for an Architect agent, but not as necessary if the agent is capable of generating its own dialogue acts, since such a model can simply use its own predictions as input in an interactive scenario.

In this section, we briefly explore how well we can identify dialogue acts in human Architect utterances. Formally, we define the task of dialogue act classification as follows: given an **A** utterance and the game context leading up to it, the task is to classify the given utterance as any number of Architect dialogue acts, as defined in Section 4.2. While we leave the question of how to use a dialogue act classifier in downstream Builder models to future work, for completeness, we present a simple baseline model and results on this task below.

4.3.1 Model

Figure 4.4 shows our model architecture. Our simple model for this task only considers the game history up to and including the Architect utterance to be classified. We embed the truncated game history as a sequence of tokens using a GRU [48] and feed the resulting context vector through a classifier, implemented as a series of feedforward layers of decreasing sizes, with a final dimensionality equal to the number of Architect dialogue act classes to be predicted. A sigmoid is applied to the final layer to produce the predictions.

Game history encoder We view the game history as a non-empty sequence of previous utterances (by both players), possibly interleaved with sequences of actions that were taken by **B** in earlier turns of the game. Similar to Narayan-Chen et al. (2019) [39], we encode the last 8 utterances in the dialogue history as a sequence of tokens in which each player’s

utterances are contained within speaker-specific start and end tokens. Following Jayannavar et al. (2020) [38], we also represent **B**’s prior actions naively as tokens that capture the action type (placement or removal) and block color (e.g. as “`builder_putdown_red`”), e.g.:

```
<A> great so then on the bottom of that green , make another 2 x 1 </A>
builder_putdown_green builder_putdown_green <B> like that ? </B> <A> sorry ,
other isde </A> builder_pickup_green builder_pickup_green builder_putdown_green
builder_putdown_green <A> great , now do another one </A>
```

The $2 \times 6 = 12$ action tokens as well as the speaker tokens are encoded using 300-dimensional random vectors, while all other tokens are encoded as 300-dimensional pre-trained GloVe word embeddings [46]. However, unlike Jayannavar et al. (2020), we allow these embeddings to be finetuned during training. The token embeddings are passed through a unidirectional 2-layer GRU to produce a 300-dim embedding of the dialogue history in the GRU’s final hidden state.

Dialogue act classifier The 300-dim embedding of the dialogue history is fed into a 2-layer feedforward network with ReLU. The linear layers reduce the dimensionality of the input embedding in equal increments, down to a 6-dim (for the original set of dialogue acts) or a 5-dim (for the simplified set) output vector. The final vector is passed through a sigmoid, then thresholded with a value of 0.5, to produce the output prediction.

4.3.2 Experimental Setup

Data After employing the data cleaning process described in the previous chapter (Section 3.7), our training, test and dev splits contain 6,422, 2,855, and 2,251 Architect utterances to be classified.

Training We trained for a maximum of 100 epochs using the Adam optimizer [63] with a learning rate of 0.0001 (but our best models converged after no more than 10 epochs). We use a dropout value of 0.5 for the game history encoder, but do not use dropout in the dialogue act classifier. During training, we minimize binary cross-entropy loss between the predicted and ground truth dialogue acts. We stop training early when loss on the held-out development set had increased monotonically for ten epochs.

Dialogue act	development			test		
	p	r	F1	p	r	F1
Instruct B	92.1	88.9	90.5	92.3	89.2	90.7
Describe Target	84.5	57.9	68.7	74.2	55.1	63.2
Answer question	83.5	81.9	82.7	73.4	77.2	75.3
Verification	85.3	85.5	85.4	86.5	87.5	87.0
Clarification/Correction	66.7	73.9	70.1	74.6	66.8	70.5
Other	76.1	70.8	73.4	66.3	68.5	67.4
Micro average	85.0	82.0	83.5	84.4	80.4	82.3

Table 4.3: Classification results for the full set of Architect dialogue acts on the development and test sets.

Dialogue act	development			test		
	p	r	F1	p	r	F1
Instruct B /Describe Target	92.0	93.1	92.6	92.4	91.4	91.9
Answer question	84.9	83.8	84.3	75.9	78.3	77.1
Verification	87.0	83.9	85.4	88.0	87.1	87.5
Clarification/Correction	70.6	72.1	71.3	76.5	61.9	68.5
Other	73.0	69.4	71.2	63.2	71.3	67.0
Micro average	86.5	86.4	86.4	86.4	84.1	85.2

Table 4.4: Classification results for the simplified set of Architect dialogue acts on the development and test sets.

4.3.3 Results

We evaluate the performance of each Architect dialogue act individually, and report precision, recall, and F1 scores for each class. We also compute micro averaged precision, recall, and F1 scores across all classes.

Original dialogue acts Dialogue act classification results for the original set of Architect dialogue acts are in Table 4.3. Just by processing the dialogue history, the simple model recognizes instructions, which are most common in our data, with reasonable performance. However, it struggles with other types of dialogue acts, particularly descriptions.

Simplified dialogue acts Dialogue act classification results for the simplified set of Architect dialogue acts, where *Instruct B* and *Describe Target* are merged, are in Table 4.4. By merging these categories, this model enjoys increased performance, particularly on recall,

on the new category while maintaining similar performance on all other categories.

4.3.4 Qualitative Analysis

Using the simplified set of Architect dialogue acts, we compare our model’s predictions to ground truth dialogue act annotations on selected samples from the development set. For each example, we provide a snippet of the dialogue history and show the Architect utterance to be classified in bolded text.

Example 4.1. When trained on the simplified set of dialogue acts, the model is able to classify utterances that incorporate multiple dialogue acts. For instance, the model is able to correctly classify the following utterance that answers a question with a verification as well as provides another instruction/description:

*A: now we must create the bell. please start by extending 4 orange blocks down from the middle purple block, as if it were hanging
(B places 3 orange blocks)*

B: like this?

A: yes. now we need 3 blocks on both sides of the bottom three blocks so that is looks like a 3x3 square hanging off of the purple row

model output: *AVI*

ground truth: *AVI*

Example 4.2. There are some instances where the model drops categories that are included in the ground truth annotation. This is particularly noticeable when an utterance contains both a verification and a clarification, e.g.:

B: orange?

(B places 3 orange blocks)

A: yes, orange. sorry

model output: *AC*

ground truth: *VAC*

Example 4.3. Sometimes, the distinction between instructions and corrections can be unclear, and this is reflected in noisy annotations. In the following example, **A** provides a response to **B** who asks for clarification. This is annotated as an answer to **B**’s question that contains a follow-up instruction, but could also be reasonably seen as a clarification to **A**’s previous instruction (which our model instead predicts):

A: *hello builder. we will start with red blocks. please build a 4x4 open window so that a 2x2 space is open in the middle*

B: *flat on the ground?*

B: *anywhere?*

A: *yes flat on the ground, preferably centered in the middle*

model output: *AC*

ground truth: *AI*

Example 4.4. Another common source of noisy annotation occurs when the *Answer question* category is mistakenly dropped from the ground truth label. While our model still misses the fact that the following utterance is also an instruction, it still reasonably classifies it as an indirect answer to **B**'s question that is asked prior to the last action sequence, while the ground truth annotation does not:

A: *now i need you to put a green block on either side of the second existing block, on the ground.*

B: *so on the middle blocks?*

(B places 2 green blocks)

A: *they need to be attached to the same block, preferable to the one closer to you*

model output: *CA*

ground truth: *CI*

As noted before, there are numerous instances with noisy annotations where relevant categories are dropped from the final annotation. Despite this, qualitative analysis shows that our model is still able to output reasonable predictions for these cases.

4.4 NEXT DIALOGUE ACT PREDICTION

Having demonstrated preliminary results on classifying dialogue acts for past Architect utterances, we now turn our focus to predicting the set of Architect dialogue acts that should occur in the next utterance. This is a useful ability for Architect agents, as it allows them to plan what types of information they should convey next and use that information as additional conditioning context when generating their next utterance. The additional layer of planning can also provide an interpretable window into the agent's otherwise rather inscrutable generation process. But before we can integrate dialogue act information into end-to-end Architect agents, we first need to examine how well models can predict Architect dialogue acts in isolation.

This is the task we explore in this section. Formally, we define the Architect dialogue act prediction task as follows: given access to the entire game state context leading up to a

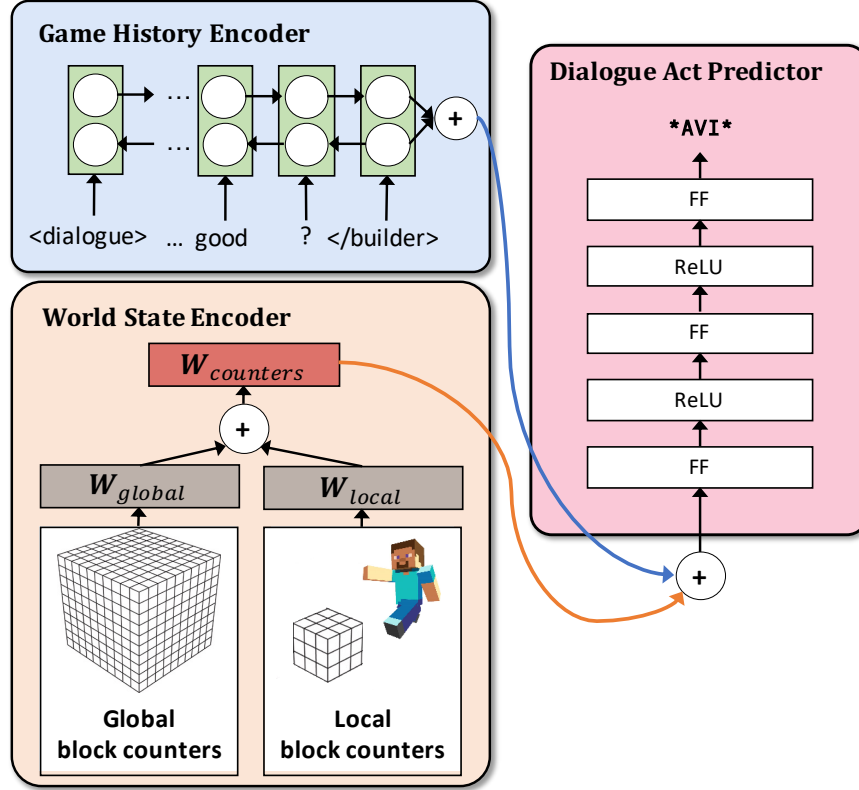


Figure 4.5: The Architect dialogue act predictor model.

certain point in a human-human game at which the human Architect **A** spoke next, we aim to generate a set of Architect dialogue acts that can reasonably follow that context. Dialogue act prediction is more challenging than classification, since multiple sets of dialogue acts can be valid for a single given input; furthermore, it requires the ability to process the world state in order to know when to issue a verification (when *Built* is correct) or a correction (when it is not). Below, we present a simple baseline model and results on this task.

4.4.1 Model

Similar to Architect utterance generation, dialogue act prediction requires the ability to simultaneously process the game history and compare the *Built* and *Target* world states in order to generate appropriate responses. For this reason, our baseline model for this task, shown in Figure 4.5, is based heavily on the block counters utterance generation model of Chapter 3. Below, we describe how we modify this architecture to instead generate Architect dialogue acts.

Game history encoder Our game history encoder is similar to that of the dialogue act classifier in the previous section. We encode the entire dialogue history as a sequence of tokens in which each player’s utterances are contained within speaker-specific start and end tokens. We also represent **B**’s prior actions using tokens that capture the action type and block color of each **B** action in the history. The $2 \times 6 = 12$ action tokens as well as the speaker tokens are encoded using 300-dimensional random vectors, while all other tokens are encoded as 300-dimensional pre-trained GloVe word embeddings [46] and finetuned during training. The token embeddings are passed through a bidirectional 1-layer GRU [48] to produce a 300-dim embedding of the dialogue history in the GRU’s final hidden state.

Block counters world state encoder In order to decide what type of information to convey next, a Architect model that predicts dialogue acts needs to also be able to process the world state by comparing *Built* and *Target*. In Section 3.4, we explored a simple way to summarize world state information for Architect utterance models via a block counters representation that recorded the colors of blocks to be placed, removed, and feasibly placed next. Here, we re-use the block counters encoder to produce an embedding of the world state for our model that predicts dialogue acts. In this model, the global counters are embedded separately into an 18-dimensional vector via a single feedforward layer, and the local counters are embedded into a 150-dimensional vector using another feedforward layer. The resulting embeddings are concatenated to produce a 168-dimensional final world state encoding.

Dialogue act predictor The 300-dim embedding of the dialogue history is concatenated with the 168-dim final world state encoding and fed into a 3-layer feedforward network with ReLU. The linear layers reduce the dimensionality of the input embedding in equal increments down to a 5-dim output vector corresponding to the simplified set of Architect dialogue acts. The final vector is passed through a sigmoid, then thresholded with a value of 0.5, to produce the output prediction.

4.4.2 Experimental Setup

Data We use the same data splits as those of the dialogue act classification task. After data cleaning, our training, test and dev splits contain 6,422, 2,855, and 2,251 samples.

Training We trained for a maximum of 40 epochs using the Adam optimizer [63] with a learning rate of 0.0001 (but our best models converged after no more than 12 epochs). During training, we minimize binary cross-entropy loss, summed across all classes, between

Dialogue act	development			test		
	p	r	F1	p	r	F1
Instruct B /Describe Target	75.0	86.2	80.2	74.6	86.3	80.0
Answer question	70.3	89.5	78.8	60.4	87.9	71.6
Verification	57.7	58.2	57.9	58.9	53.7	56.2
Clarification/Correction	42.8	37.9	40.2	48.7	37.9	42.6
Other	45.5	10.5	17.0	61.9	21.9	32.4
Micro average	67.5	71.2	69.3	67.3	69.9	68.6

Table 4.5: Prediction results for the simplified set of Architect dialogue acts on the development and test sets.

the predicted and ground truth dialogue acts. Because of the class imbalance in the training data (instructions are far more frequent than any other class), we also found it useful to weight each class in the loss function by the ratio of negative to positive samples of that class in the training data, thus accruing larger losses per sample for the classes that were less frequently seen.¹ However, using this weighted loss as a stopping criteria resulted in model training stopping too early. Therefore, we instead stop training early when the micro-averaged F1 score on the held-out development set had decreased monotonically for five epochs.

4.4.3 Results

Using the simplified set of Architect dialogue acts, we evaluate the performance of each Architect dialogue act individually, and report precision, recall, and F1 scores for each class. We also report the micro-averaged precision, recall, and F1 scores across classes.

The results in Table 4.5 highlight how much more challenging dialogue act prediction is compared to the classification task. Particularly, the model struggles to identify when to issue verifications and clarifications/corrections, suggesting that the block counters world representation does not provide a strong enough signal about how *Built* deviates from *Target*.

4.4.4 Qualitative Analysis

In order to better understand our model’s performance as well as analyze the effect of annotation noise, we perform a qualitative analysis of our model’s predictions on selected samples from the development set. For each example, we first provide a snippet of the

¹The resulting weights used were: *I/D*: 0.5546, *A*: 6.8701, *V*: 3.6168, *C*: 4.234, *O*: 14.2905.

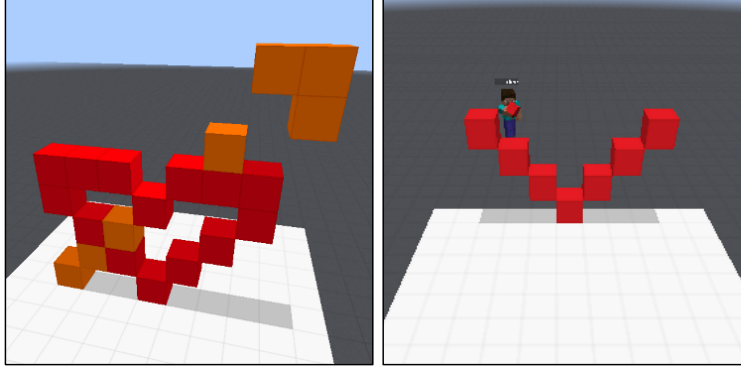


Figure 4.6: *Target* (left) and *Built* (right) structures for Example 4.5.

dialogue history, then show our model’s prediction and ground truth annotation for the dialogue acts that should follow that context. For completeness, we also include the next human Architect utterance itself (which is not used anywhere during training).

Example 4.5. In the following example (Figure 4.6), after mistakenly building the structure flat on the ground, **B** executes a long sequence of actions following an **A** correction and asks **A** whether the fix was correctly executed. Our model correctly predicts that it should respond with a verification, and even goes further than the human **A**’s original response by also issuing another instruction (which is still appropriate, given that *Built* is not yet finished at this point):

A: *Sorry, this is going to look sort of like the bell with a vertical design*

B: *oh, ok!*

(B removes 5 red blocks)

A: *No worries*

(B removes 7 red blocks and places 11 red blocks)

B: *like that?*

model output: *VAI*

ground truth: *VA*

next A utterance: *Yup,*

Example 4.6. Figure 4.7 illustrates the beginning of a dialogue. With a friendly opener, **B** asks **A** for a description of the *Target* to be built. Our model produces a reasonable response, predicting that an **A** instruction should come next. However, the ground truth annotation notes that the next utterance should also be a response to **B**’s initial question, which our model ignores (possibly due to the inconsistent labeling of the *Answer question* dialogue act in the training data):

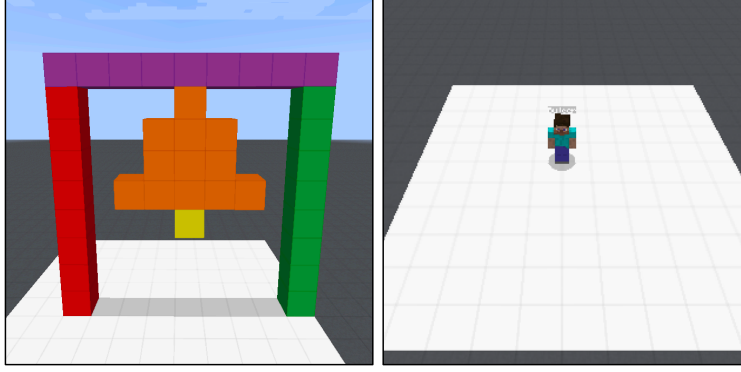


Figure 4.7: *Target* (left) and *Built* (right) structures for Example 4.6.

B: *hello. what are we building this time?*

model output: *I*

ground truth: *AI*

next A utterance: *hello builder, i will tell you this. it appears we are creating a belltower. but first i will start with step by step instructions. we will start with green blocks*

Example 4.7. In Figure 4.8, **B** has successfully completed building the green pillar in *Target*. **A** then uses the newly created pillar as a reference for another to be built. It is clear simply from the dialogue context that **A**’s initial instruction was ambiguous, both from **B**’s clarification question as well as **A**’s resulting apology (“*sorry, my mistake*”). The model picks up on this and issues a clarification/correction, but still leaves it underspecified, while the human **A** also provides a follow-up instruction to rectify the mistake:

A: *on the 8th block to the left of the green block, please create the same structure with red blocks*

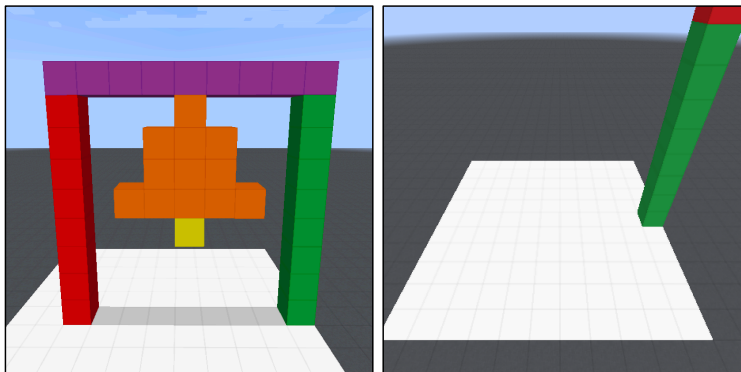


Figure 4.8: *Target* (left) and *Built* (right) structures for Example 4.7.

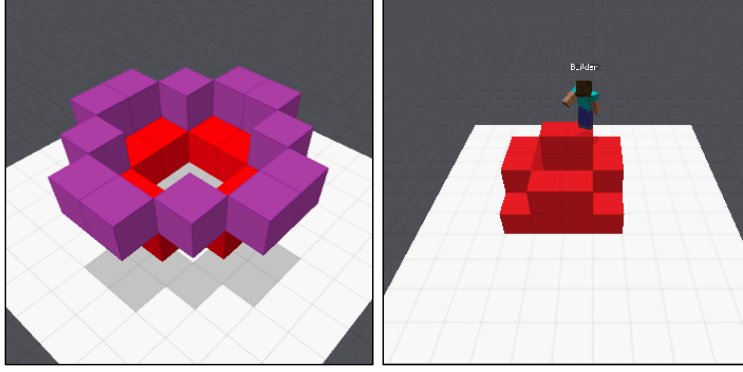


Figure 4.9: *Target* (left) and *Built* (right) structures for Example 4.8.

B: *on top the 8th block?*

A: *sorry, my mistake*

(B places a red block)

model output: *C*

ground truth: *CI*

next A utterance: *it should start on the ground*

Example 4.8. As with the dialogue act classification task, instructions and clarifications can easily be confused in noisy annotations. In the example below (Figure 4.9), **A** provides yet another confusing initial instruction, to which **B** asks for clarification by requesting more lower-level instructions. In this case, our model reasonably predicts that the next **A** utterance should answer **B**’s question with a clarification, while the ground truth annotation just labels this as an answer that follows up with an instruction:

A: *there is a 3rd layer that is floating, so please made a 6x6 square that extends upwards of 2 blocks to support the 3rd layer*

A: *these will use purple blocks*

B: *can you give me an instruction for a single purple block first?*

model output: *AC*

ground truth: *AI*

next A utterance: *how about this, please make a 6x6 purple ring around the first layer of the red square*

Example 4.9. Since standalone verifications can appear arbitrarily after **B** has finished executing a sequence of actions correctly, our model sometimes chooses to push on with further instructions while the human **A** might instead stop to provide positive feedback (and vice versa), such as in the following example (Figure 4.10):

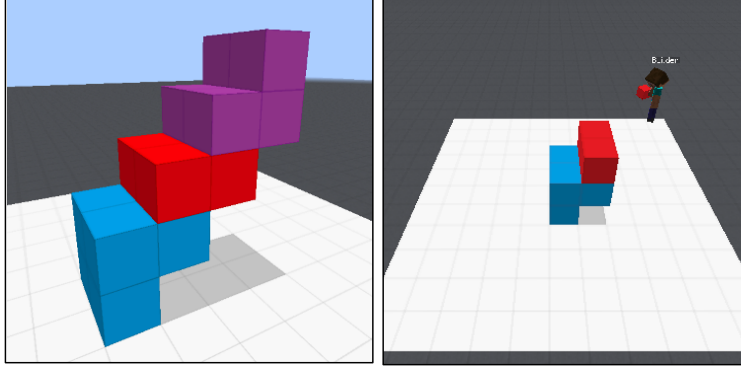


Figure 4.10: *Target* (left) and *Built* (right) structures for Example 4.9.

A: *on top of the blocks you just placed, put down two red blocks*
(B places and removes a red block, then places 2 red blocks)

model output: *I*

ground truth: *V*

next A utterance: *great*

Example 4.10. Deciding when to appropriately issue verifications rather than corrections is still hard for our model. The dialogue context alone often does not provide enough information about whether **B**'s last sequence of actions correctly follows **A**'s instruction; in the following example (Figure 4.11), the only indication that a mistake has been made includes a minimal **A** interruption and **B**'s subsequent undoing of some block placements. Here, the model must rely on comparing *Built* and *Target* to identify that a mistake has been made, but fails to do so:

A: *add yellow blocks on top and connect them across the bridge*
(B places 3 yellow blocks)

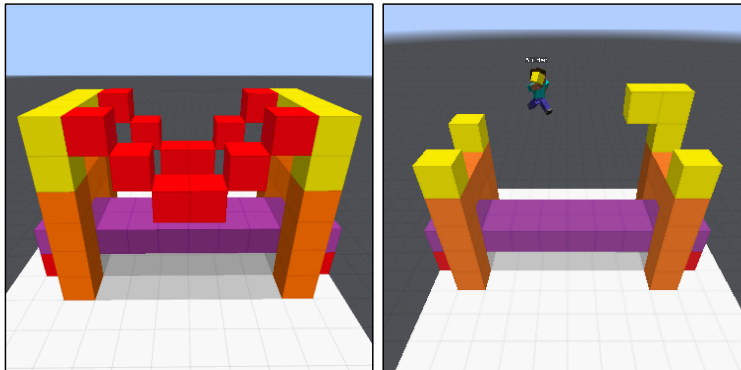


Figure 4.11: *Target* (left) and *Built* (right) structures for Example 4.10.

A: *wait*

(B removes 2 yellow blocks)

model output: *V*

ground truth: *CI*

next A utterance: *other way*

Overall, qualitative analysis reveals that while our model is penalized for dropping or adding dialogue act categories, the model’s behavior when doing so is mostly reasonable when taking into account the corresponding dialogue context and overall annotation noise. However, the mistakes the model makes in distinguishing verifications and clarifications/corrections cannot be reasoned away. Since knowing when to give appropriate feedback about *Built* with respect to *Target* is important for overall interactive task completion, future work should aim to address this problem by encoding richer representations of the world state.

4.5 DIALOGUE ACTS FOR ARCHITECT UTTERANCE GENERATION

Finally, we examine how we can leverage Architect dialogue acts to train better models for downstream tasks. In particular, our block counters utterance generation models in Chapter 3 treat utterances in a rather simple manner, representing them as no more than sequences of tokenized word embeddings that are consumed by an RNN. Here, we propose to use Architect dialogue acts to augment this representation with richer linguistic information. In this section, we explore one way of integrating Architect dialogue acts into our existing utterance generation models by using the set of dialogue acts, both ground truth as well as those predicted by our model in Section 4.4, as additional conditioning input to the utterance decoder, and investigate their effect on the resulting utterances through both quantitative and qualitative analysis.

4.5.1 Model

Our utterance generation models that leverage dialogue acts augment the original block counters model with Architect dialogue act information in the RNN decoder. To do this, we construct a multi-hot vector encoding the dialogue acts, in which dialogue acts that are enabled for a given utterance are assigned values of 1 (and others values of 0). In addition to including the original block counters encoding, we concatenate the dialogue act vector (or, optionally, an embedding of it) to the decoder’s input at every time step. For completeness, we briefly describe the entire model architecture, visualized in Figure 4.12, below.

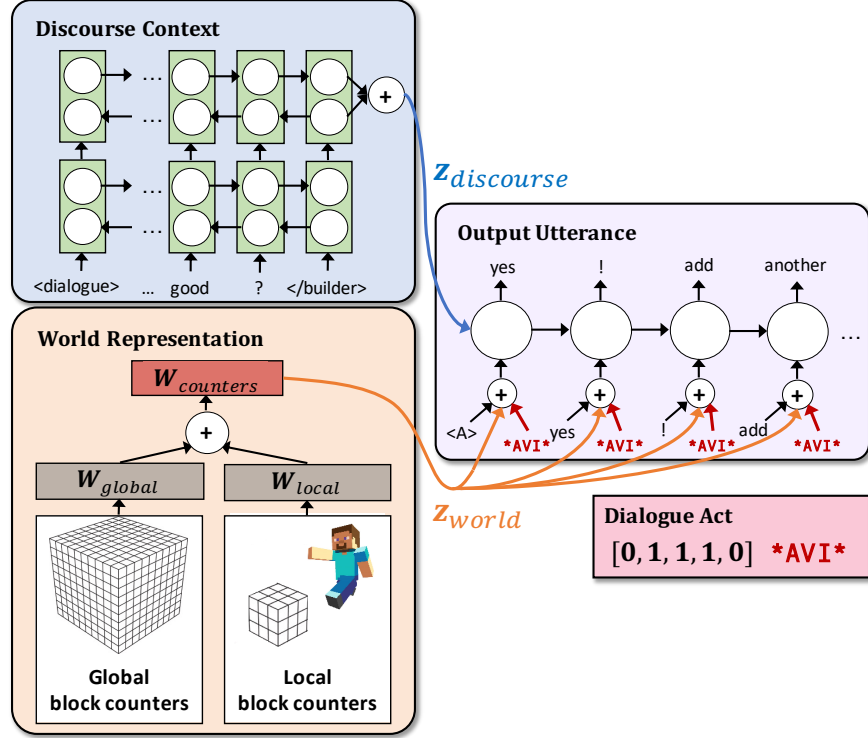


Figure 4.12: The Architect block counters utterance model conditioned on dialogue acts.

Dialogue history encoder Our dialogue history encoder remains the same as that of the original block counters model described in Section 3.3. We encode the entire dialogue history as a sequence of tokens in which each player’s utterances are contained within speaker-specific start and end tokens, which are fed through a word embedding layer and subsequently passed through a bidirectional RNN [44] to produce an embedding of the entire dialogue history in the encoder RNN’s final hidden state.

Output utterance decoder As before, the output utterance is generated by a decoder RNN conditioned on the discourse context. In standard fashion, the final hidden state of the encoder RNN is used to initialize the hidden state of the decoder RNN.

Block counters world state encoder The block counters encoder used for this model remains the same as that of the updated block counters v2 model described in Section 3.7. We found the best results by embedding the global and local counter vectors through separate encoders, then concatenating the resulting embeddings and feeding it through a single fully-connected layer to produce the final block counters embedding. The resulting embedding is then concatenated to the word embedding vector that is fed into the output utterance decoder at each time step.

Multi-hot dialogue act encoding Since multiple Architect dialogue acts can be active at a time, we represent the set of dialogue acts that should be expressed next by the decoded utterance as a 5-dimensional multi-hot vector, where each bit corresponds to one of the simplified set of Architect dialogue acts. This vector is optionally embedded into another vector of the same size using a single feedforward layer with ReLU. We add this information to the output utterance decoder by simply concatenating it to the decoder’s input at every time step. In our final model, at every time step, the decoder RNN processes the word embedding of the previously generated token, the final block counters embedding, and the dialogue act representation as a concatenated input vector.

4.5.2 Experimental Setup

Data We use the same data splits as those used by the block counters v2 model. After data cleaning, our training, test and dev splits contain 6,422, 2,855, and 2,251 samples.

Hyperparameters We perform grid searches over hyperparameters for two model variants: one trained using ground truth dialogue act annotations, and another trained using the predicted dialogue acts generated by thresholding the outputs of our model in Section 4.4. We use 300-dimensional bidirectional Gated Recurrent Units (GRUs) [48] for all encoder RNN modules, where the sum of the two final hidden states in either direction constitutes the final encoding of dialogue history (used to initialize the decoder). Both RNNs use 300-dimensional pretrained GloVe word embeddings [46]. All linear layers were initialized using Xavier initialization [64]. All models featured single-layer decoder RNNs and were trained with dropout of 0.5 for both the RNNs and the block counter embedding layers. For models that used predicted dialogue acts, our dialogue act predictor model follows the architecture outlined in Section 4.4. See Appendix D.4 for detailed hyperparameter configurations.

Training We trained for a maximum of 40 epochs using the Adam optimizer [63] with a learning rate of 0.0001. During training, we minimize the sum of the cross-entropy losses between each predicted and ground truth token. We stop training early when perplexity on the held-out development set had increased monotonically for two epochs. When training using predicted dialogue acts, we do not finetune the dialogue act predictor model itself, and instead only use its output predictions.

Decoding For all models, we use constrained beam search decoding, subject to the constraints defined in Section 3.7, to generate the utterance with the maximum log-likelihood

score according to our model normalized by utterance length (beam size = 10). In order to promote diversity of generated utterances, we use a γ penalty [55] of $\gamma = 0.8$. For the model trained using ground truth dialogue act labels, we experiment with decoding both with ground truth as well as predicted dialogue acts at test time. For the model trained using predicted dialogue acts, we only use predicted dialogue acts for decoding at test time.

4.5.3 Results

With the block counters v2 model (described in Section 3.7) as our baseline, we report the performance of all our models using BLEU [41] and term-specific precision and recall on the development set in Table 4.6 and on the test set in Table 4.7. Additionally, on the development set, we perform a more fine-grained analysis by examining model performance on samples categorized by their annotated dialogue acts.

The model that uses ground truth dialogue acts for both training and decoding enjoys the biggest performance boost in BLEU and overall term-specific scores. Specifically, dialogue term-specific scores are noticeably better than those of the model without dialogue act information at the cost of a slight drop in color term scores (though, aided by constrained decoding, these still stay within a reasonable ballpark), and these trends hold on the unseen test set. The benefits are less clear with a switch to using predicted dialogue acts, both when applied to the model trained on ground truth dialogue acts as well as one trained on the predicted acts themselves. Both variants enjoy better performance on dialogue terms, though not nearly to the extent of the model trained and decoded using ground truth acts, and there is a larger negative effect on color term scores. However, overall term-specific recall scores for both models still show slight improvement over the model without dialogue acts, with a minimal improvement (if at all) to BLEU score.

Table 4.6 also provides a breakdown of model performance on the development set by comparing them against human utterances with specific dialogue acts. These clearly highlight that the baseline block counters model without dialogue acts optimizes performance on instructions, boasting the highest color term scores for this category; however, this model’s performance is not particularly noteworthy for the remaining dialogue acts. On the other hand, models that leverage dialogue act information show clear improvement on all terms for non-instruction acts while maintaining reasonable performance on instructions. These models shine when evaluated specifically on *Answer question* and *Verification* dialogue acts, as they generate appropriate dialogue terms for these types of utterances much more reliably. However, regardless of model, performance on the *Clarification/Correction* category is poor; besides the fact that this umbrella category is the most noisily annotated, improving perfor-

		BLEU				Precision / Recall							
		B-1	B-2	B-3	B-4	all keywords		colors	spatial		dialogue		
baseline	all	17.7	8.9	5.1	3.1	15.5 /	15.0	27.2 /	28.9	11.8 /	9.4	18.0 /	21.4
	I	17.2	8.8	5.1	3.2	18.5 /	14.8	35.0 /	29.9	15.2 /	9.9	18.7 /	20.4
	A	11.6	5.0	2.5	1.4	15.6 /	14.5	9.0 /	13.0	6.1 /	4.8	25.9 /	26.0
	V	12.1	5.3	2.5	1.3	15.3 /	16.7	15.2 /	24.2	5.2 /	6.8	26.9 /	24.4
	C	11.9	5.4	2.6	1.2	8.8 /	9.4	13.6 /	21.8	8.2 /	5.8	6.6 /	10.6
	O	3.7	.59	0.0	0.0	3.7 /	11.7	1.4 /	16.7	0.0 /	0.0	9.5 /	15.7
gold train/test	all	22.7	11.8	6.7	4.1	19.5 /	18.8	28.7 /	27.4	11.1 /	11.0	24.6 /	29.3
	I	23.5	12.6	7.4	4.6	18.9 /	17.7	29.4 /	28.8	11.5 /	11.8	22.1 /	24.6
	A	17.7	8.4	3.7	1.6	25.0 /	23.5	24.3 /	23.4	9.8 /	7.0	35.5 /	39.9
	V	19.5	10.0	5.6	3.4	30.6 /	28.0	23.2 /	21.0	13.4 /	11.4	41.7 /	43.8
	C	15.3	7.4	3.5	1.5	9.6 /	12.2	11.8 /	15.1	7.7 /	6.9	9.1 /	17.8
	O	2.1	.96	0.0	0.0	15.0 /	12.7	0.0 /	0.0	0.0 /	0.0	16.9 /	18.2
+ pred. test	all	18.9	9.6	5.3	3.3	15.9 /	16.0	24.2 /	24.6	9.0 /	9.6	19.5 /	23.1
	I	20.4	10.8	6.4	4.0	18.6 /	15.5	30.2 /	25.5	11.8 /	10.5	20.0 /	21.1
	A	10.4	4.5	2.0	1.0	21.0 /	18.6	4.9 /	3.9	5.6 /	3.8	32.8 /	35.7
	V	12.7	5.8	3.1	1.9	16.2 /	19.1	9.5 /	14.5	4.4 /	7.2	31.1 /	29.1
	C	12.5	5.7	2.7	1.2	8.1 /	10.0	8.9 /	12.6	5.6 /	5.6	6.0 /	11.0
	O	4.8	2.5	1.6	.81	3.0 /	9.4	1.5 /	16.7	0.0 /	0.0	6.5 /	10.7
pred. train/test	all	18.2	9.2	5.4	3.4	16.2 /	15.9	26.5 /	26.4	9.7 /	9.2	19.6 /	23.2
	I	17.9	9.3	5.6	3.5	19.0 /	15.4	33.9 /	27.2	13.0 /	10.1	19.8 /	21.4
	A	11.4	5.3	2.9	1.8	15.8 /	16.0	9.1 /	11.7	2.6 /	2.7	27.4 /	28.4
	V	13.4	6.4	3.6	2.3	18.4 /	19.6	13.4 /	17.7	5.2 /	7.3	31.8 /	29.6
	C	11.3	4.7	2.3	1.2	7.9 /	9.8	9.2 /	16.8	3.9 /	3.9	6.6 /	11.0
	O	5.3	2.2	1.3	0.0	4.3 /	11.1	0.0 /	0.0	0.0 /	0.0	11.2 /	15.7

Table 4.6: BLEU and term-specific precision and recall scores for the ablation study on the development set across all dialogue acts (first row of each section) as well as for only those samples annotated as a specific dialogue act type. In order, we report results from the original block counters v2 model (section 1); a model that uses ground truth dialogue acts for both training and decoding (section 2); that same model but with predicted dialogue acts during decoding (section 3); and a model that uses predicted dialogue acts for both training and decoding (section 4).

Model	BLEU				Precision / Recall					
	B-1	B-2	B-3	B-4	all keywords	colors	spatial	dialogue		
baseline	17.4	9.2	5.6	3.6	15.0 / 15.0	25.9 / 30.2	10.8 / 8.4	16.6 / 21.0		
+ DA										
gold	21.9	11.5	6.7	4.2	18.3 / 19.5	25.6 / 29.4	10.7 / 10.8	23.3 / 29.9		
pred. test	18.3	9.7	5.7	3.6	14.9 / 16.4	22.3 / 25.9	9.1 / 10.1	17.7 / 22.9		
predicted	17.2	8.7	5.5	3.7	15.0 / 16.1	24.3 / 26.7	9.1 / 8.9	18.0 / 22.5		

Table 4.7: BLEU and term-specific precision and recall scores on the test set. In order, we report results from the original block counters v2 model (row 1); a model that uses ground truth dialogue acts for both training and decoding (row 3); that same model but with predicted dialogue acts during decoding (row 4); and a model that uses predicted dialogue acts for both training and decoding (row 5).

mance on this category requires that models can identify mistakes and describe how to rectify them, which is clearly outside the capacity of the current block counters representation.

4.5.4 Qualitative Analysis

Here, we qualitatively compare the outputs of our model variants on samples from the development set. For each example, we provide a snippet of the dialogue history and a visualization of *Built* and *Target*, then list both the ground truth and predicted dialogue acts for the following **A** utterance. We then show each of our models’ outputs along with the actual human **A** utterance. Though we do not perform a full human evaluation, we highlight here some of the new and desirable features exhibited by models that leverage dialogue acts, as well as showcase some interesting scenarios that are challenging to evaluate.

Example 4.11. Figure 4.13 shows the end of a game where a structure has been completed successfully. Without dialogue acts, our previous model struggles to identify when instructions no longer need to be issued, puttering on with incorrect instructions. On the other hand, all of our models that use dialogue acts pick up on the strong cue of an end-of-game chit-chat utterance and respond appropriately (and with a smile):

A: *builder, we have done it*

B: *good job!*

ground truth DA: *0*

predicted DA: *0*

block counters v2: *now we ’ll do the same thing on the other side*

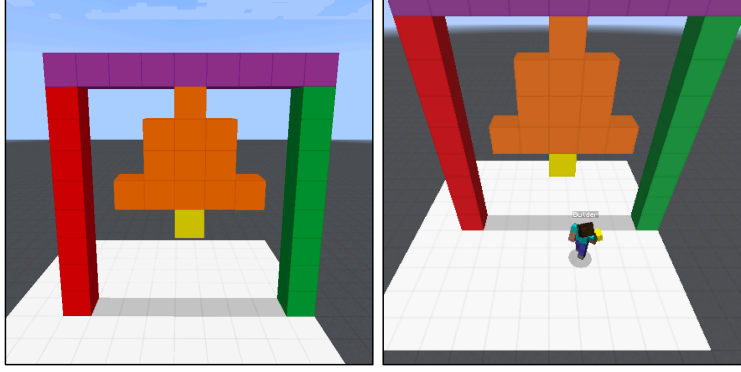


Figure 4.13: *Target* (left) and *Built* (right) structures for Example 4.11.

gold train/test: *good job :)*

gold train/pred. test: *good job :)*

predicted train/test: *looks good ! :)*

human utterance: *you too builder! you are a master craftsman/woman*

Example 4.12. In the following example (Figure 4.14), **A** has finished instructing **B** to build a first temporary layer needed to construct the floating *Target*. Moving on to the next set of instructions, **A** primes **B** by describing how the first layer will be used. Possibly because the block counters indicate that a large number of temporary blocks need to be removed, our original model issues a correction that is both incorrect and inappropriate given the dialogue context. However, knowing that the next utterance should be an instruction, our other models generate more appropriate responses:

A: *we will be removing this first layer in the future, but we need to build on top of it first.*

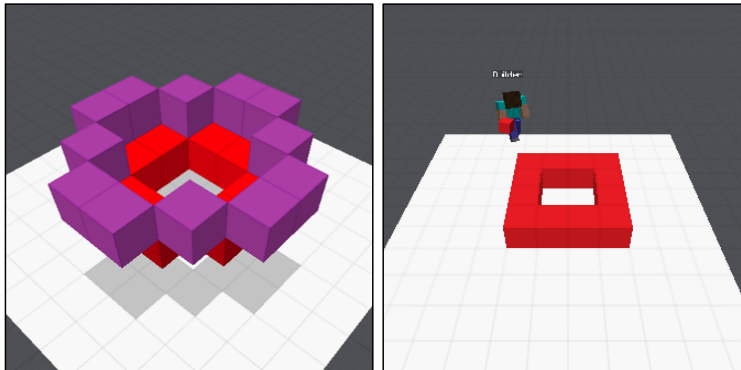


Figure 4.14: *Target* (left) and *Built* (right) structures for Example 4.12.

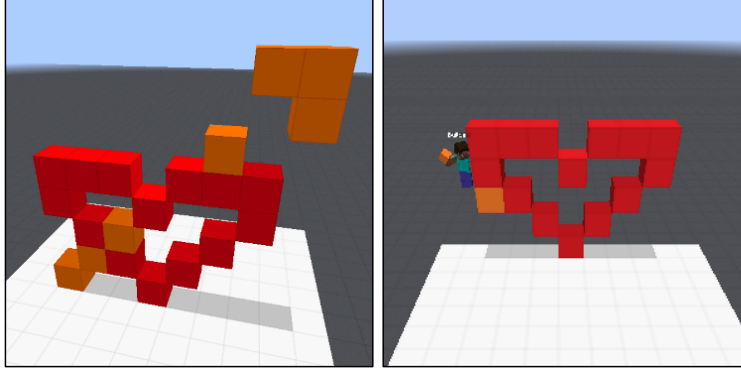


Figure 4.15: *Target* (left) and *Built* (right) structures for Example 4.13.

ground truth DA: *I*

predicted DA: *I*

block counters v2: *sorry , it should be on the ground*

gold train/test: *now add two more red blocks on top of the red block*

gold train/pred. test: *place a red block on top of the red block*

predicted train/test: *on top of the red block , place a red block on top of the red block*

human utterance: *please place 2 red blocks on top of each side of the square, leaving the corners empty*

Example 4.13. In the next example (Figure 4.15), **B** has just finished building the heart structure, so **A** begins to instruct them to build the orange arrow that pierces it by describing the start of a diagonal line of blocks. **B** misinterprets **A** and places an orange block in the wrong location. Both the model without dialogue acts and the models that use the incorrectly predicted dialogue acts erroneously generate verifications. In contrast, the model that uses ground truth dialogue acts generates a correction that is surprisingly appropriate:

A: *Now we are going to build the arrow. Under the furthest right block, start a diagonal line of orange blocks.*

B: *under it?*

(B places an orange block)

B: *here?*

ground truth DA: *AC*

predicted DA: *VAC*

block counters v2: *yup !*

gold train/test: *sorry , it should be on the ground*

gold train/pred. test: *yes !*

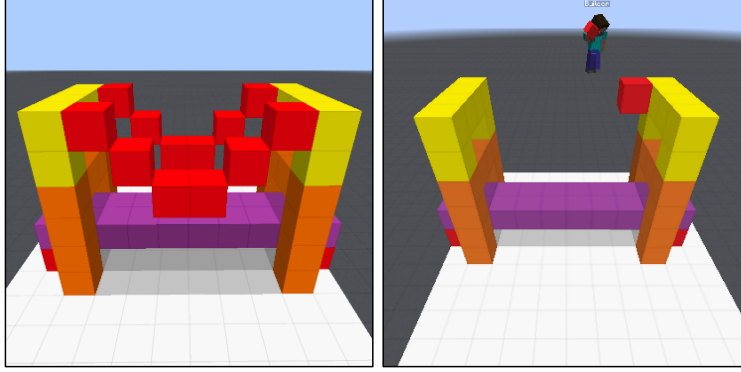


Figure 4.16: *Target* (left) and *Built* (right) structures for Example 4.14.

predicted train/test: *yes*

human utterance: *Sorry, from the ground*

Example 4.14. In the example below (Figure 4.16), **A** starts to instruct **B** on how to build the red suspension cables for the bridge. **B** interprets **A**’s instruction correctly and begins to execute it. However, due to crossed wires, **A** thinks their instruction isn’t sufficiently clear and proceeds to issue a correction, apologizing for the ambiguity. (Indeed, just after this point, **B** asks, “*here?*”, and **A** responds, “*oh yeah great.*”) The original block counters model is completely oblivious to this context and continues issuing (incorrect) instructions. The model that uses the ground truth dialogue act indeed issues a correction, though the correction itself is incorrect, since no correction was actually needed here. Interestingly, the models that use predicted dialogue acts respond most appropriately, having predicted that a verification should follow:

A: *start with red blocks*

A: *place them next to the top yellow blocks on the posts so they are on the inside of the bridge*

(B places a red block)

ground truth DA: **C**

predicted DA: **V**

block counters v2: *now place a red block on top of the middle red block*

gold train/test: *other side sorry*

gold train/pred. test: *yeah like that*

predicted train/test: *great*

human utterance: *that wasn’t clear sorry*

Qualitative analysis of our models reveal that, when correct, predicted dialogue act information can be very valuable context for utterance generation models. As our Architect

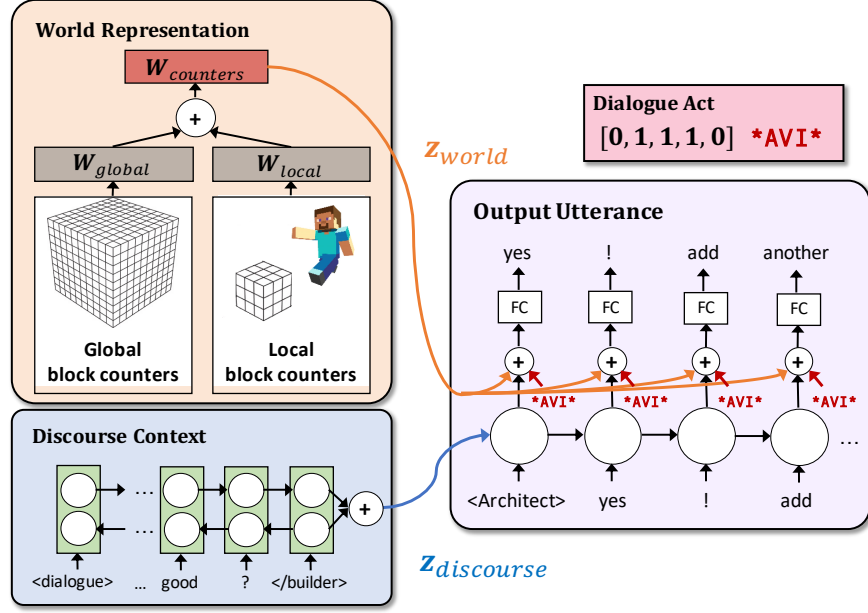


Figure 4.17: The modified block counters v2 model architecture using both oracle next **B** actions and ground truth **A** dialogue acts. The local block counters are computed using the next **B** action as a reference (instead of the last **B** action).

models continue to improve on utterance correctness, as well as learn to generate higher-level concepts and corrections more appropriately, the benefits of using dialogue acts in these models will only become even more pronounced.

4.6 COMBINING DIALOGUE ACTS WITH ORACLE NEXT ACTIONS

In Section 3.8 of the previous chapter, we showed that adding oracle information by way of encoding **B**'s next action, as opposed to **B**'s last action, showed modest improvement in overall results. In this section, we investigate the compound effect of injecting both ground truth dialogue acts and ground truth next **B** action information into our block counters model. We hypothesize that by decomposing our model inputs using distinct subtasks involving both next dialogue act and next action prediction, the overall performance of our model should improve as we relieve the burden to learn all aspects of the task in a fully end-to-end fashion.

To add ground truth dialogue acts to our updated block counters model that uses oracle next **B** action information (as described in Section 3.8), we simply concatenate the multi-hot dialogue act encoding to the embedded block counters representation, which is computed using the oracle next **B** action as a reference, before combining the result with the decoder

RNN’s output (see Figure 4.17). Therefore, at every time step, the decoder’s output is concatenated with representations of the dialogue act and the global and local block counters, embedded via a feedforward network, then used to generate the next output token using a final linear layer.

4.6.1 Experimental Setup

We use the same data splits as those used by the block counters v2 model. After data cleaning, our training, test and dev splits contain 6,422, 2,855, and 2,251 samples. We compare a block counters model that uses both ground truth **A** dialogue acts and oracle next **B** actions to several previous baselines: the block counters v2 model (Section 3.7), a model that only uses oracle next **B** actions (Section 3.8), and a model that only uses ground truth dialogue act annotations (Section 4.5). All models use 1-layer bidirectional encoder RNNs for the dialogue history (with the exception of the model that only uses dialogue acts, which uses a 2-layer encoder), train with a dropout of 0.5, and use constrained decoding during generation. Both models that use dialogue acts also encode **B** action tokens in the dialogue history and enable trainable word embeddings. See Appendix D.5 for detailed hyperparameter configurations.

4.6.2 Results

Tables 4.8 and 4.9 show the results of our new model that utilizes both ground truth **A** dialogue acts and **B** next actions on the validation and test sets, respectively. Compared to models that use none or only one of these additional inputs, our new model performs better nearly across the board. Specifically, compared to the model variant that utilizes only **B** next actions, precision and recall on dialogue terms are much stronger. On the other hand, compared to the model variant that utilizes only **A** dialogue acts, performance on color and spatial terms improves dramatically while maintaining comparable performance on dialogue terms and BLEU. Our model exhibits the most marked improvement on color precision: this is possibly due to learning from cleaner categories of data, as specified by **A** dialogue acts, combined with focusing the local block counters on more relevant regions of the world, as dictated by **B** next actions.

Model	BLEU				Precision / Recall					
	B-1	B-2	B-3	B-4	all keywords	colors	spatial	dialogue		
baseline	17.7	8.9	5.1	3.1	15.5 / 15.0	27.2 / 28.9	11.8 / 9.4	18.0 / 21.4		
+ NA only	19.5	9.6	5.7	3.5	17.9 / 16.9	28.0 / 34.9	12.6 / 11.2	21.8 / 19.2		
+ DA only	22.7	11.8	6.7	4.1	19.5 / 18.8	28.7 / 27.4	11.1 / 11.0	24.6 / 29.3		
+ both	22.9	12.2	7.3	4.6	20.8 / 20.0	40.4 / 34.6	14.3 / 12.4	26.1 / 29.5		

Table 4.8: BLEU and term-specific precision and recall scores on the development set using oracle next **B** actions (NA) and ground truth dialogue acts (DA).

Model	BLEU				Precision / Recall					
	B-1	B-2	B-3	B-4	all keywords	colors	spatial	dialogue		
baseline	17.4	9.2	5.6	3.6	15.0 / 15.0	25.9 / 30.2	10.8 / 8.4	16.6 / 21.0		
+ NA only	18.6	9.3	5.5	3.5	16.8 / 17.6	23.8 / 36.0	11.8 / 11.0	19.8 / 18.8		
+ DA only	21.9	11.5	6.7	4.2	18.3 / 19.5	25.6 / 29.4	10.7 / 10.8	23.3 / 29.9		
+ both	21.9	12.0	7.5	5.0	19.1 / 19.9	35.6 / 35.1	14.6 / 12.8	23.8 / 29.1		

Table 4.9: BLEU and term-specific precision and recall scores on the test set using oracle next **B** actions (NA) and ground truth dialogue acts (DA).

4.6.3 Qualitative Analysis

Here, we analyze a few qualitative examples comparing the outputs of various models: (1) the **block counters v2** baseline (Section 3.7), (2) **NA only**: a model that uses only oracle **B** next actions (Section 3.8), (3) **DA only**: a model that utilizes only ground truth **A** dialogue acts (Section 4.5), and (4) **NA + DA**: the model described in this section that uses both ground truth **A** dialogue acts and oracle **B** next actions.

Example 4.15. In this example (Figure 4.18), **B** has just finished building a pillar of 8 green blocks, but has trouble understanding **A**’s next instructions, which describe the identical pillar of 8 red blocks on the opposing side of the board. While the model trained with **A** dialogue acts picks up that it should be issuing an instruction here, the block counters centered around **B**’s last action at the top of the green pillar have no view of the red blocks that **A** plans to describe next, leading the model to refer instead to the purple blocks that should be placed atop the green. On the other hand, the model trained with **A** dialogue acts that also uses the next **B** action correctly references the red blocks (though *bottom left corner* is not completely correct):

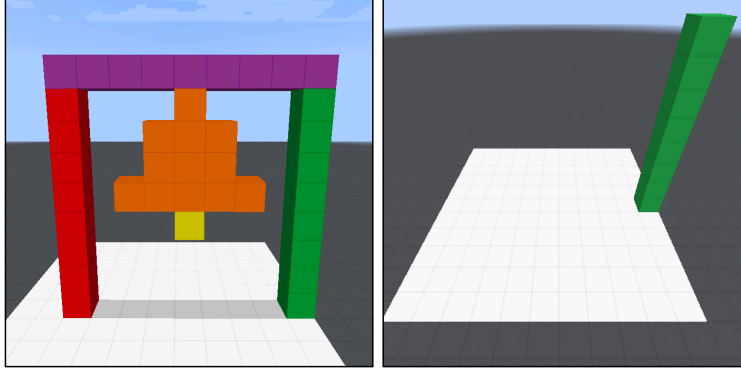


Figure 4.18: *Target* (left) and *Built* (right) structures for Example 4.15.

A: *on the 8th block to the left of the green block, please create the same structure with red blocks*

B: *on top the 8th block?*

A: *sorry, my mistake*

(B places a red block)

A: *it should start on the ground*

(B picks up a red block)

ground truth DA: **CI**

block counters v2: *yup , it should be on the ground*

NA only: *thats it*

DA only: *on the other side of the purple block , place a purple block*

NA + DA: *on the bottom left corner , place a red block on the ground*

human utterance: *i only meant 8 blocks away from the first green block, to the left*

Example 4.16. In Figure 4.19, **B** has finished removing a number of purple blocks to complete the top layer of the structure. The only steps left are to remove the bottom layer of red blocks. While the previous models that do not use **A** dialogue acts mention red blocks, they issue instructions to add more of them, rather than remove them. On the other hand, while the model that uses **A** dialogue acts expresses both a verification and an instruction, the generated instruction is vague and incorrect. While our new model does not describe this last removal step quite as well as the human does, it exhibits a step in the right direction by instructing a removal rather than a placement:

A: *we need to remove the pieces that were directly next to the old corners you just removed*

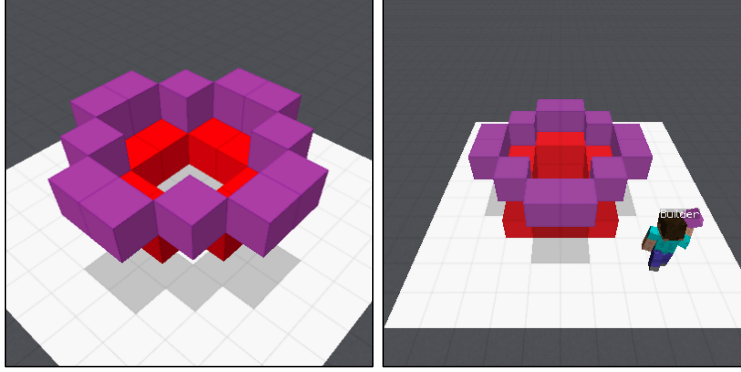


Figure 4.19: *Target* (left) and *Built* (right) structures for Example 4.16.

B: *the purple ones?*

A: *while leaving the new inner corners intact*

A: *yes*

(B removes 8 purple blocks)

ground truth DA: **IV**

block counters v2: *now we 're going to make a 3 x 3 square of red blocks on the ground*

NA only: *okay , now we 're going to make a 3 x 3 square of red blocks*

DA only: *nice . now we are going to make the same thing on the other side*

NA + DA: *okay , now remove the bottom two blocks*

human utterance: *builder, you are awesome. now all we need to do is remove the first red layer and the structure should be complete. please standby for me to inspect at the end*

Example 4.17. In Figure 4.20, following **A**'s previous instructions, **B** has finished building the sides of the red U-shape. The final step involves making the most recently-placed red blocks float by removing the red supporting blocks underneath. Only our latest model that utilizes both **A** dialogue acts and **B** next actions generates an appropriate block removal instruction, which is very similar to the human **A**'s utterance:

A: *ok then at the first and third blocks*

A: *add a red block to each side*

(B places 2 red blocks)

A: *ya good*

A: *then add a red block on top of those 2*

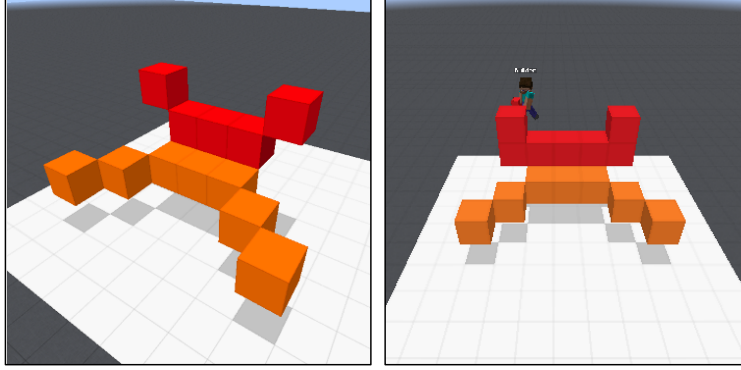


Figure 4.20: *Target* (left) and *Built* (right) structures for Example 4.17.

(B places 2 red blocks)

ground truth DA: *I*

block counters v2: *and then another block to the right of the one you just put down*

NA only: *then put another block to the right of that one*

DA only: *and then another one block to the left of the one you just put down*

NA + DA: *then get rid of the bottom red block*

human utterance: *then get rid of the red blocks underneath those*

Qualitative analysis reveals that despite sticking to the same relatively simple block counters model architecture, representing both ground truth **A** dialogue acts and oracle **B** next actions results in markedly different and promising generated outputs.

4.7 CONCLUSION

In this chapter, we explored adding richer linguistic structure to our data and models by annotating Architect utterances in our corpus with one or more of their corresponding dialogue acts, coarsely defined as instructions, descriptions, answers to questions, verifications, clarifications/corrections, and chit-chat. We investigated how well we can model that linguistic structure, either in past Architect utterances (dialogue act classification) or for future ones (dialogue act prediction), and show that even relatively simple models make reasonable predictions for these tasks. Additionally, we leverage that structure to improve existing Architect utterance models, and show that, quantitatively and qualitatively, dialogue acts provide important and useful signals that are otherwise difficult for baseline block counters models to pick up on. Furthermore, when combined with a more informed view of the world guided by oracle **B** next actions, our model shows increased improvement both quantitatively

and qualitatively, and generates utterances that are more reasonable in context.

In the next chapter, we explore how we can similarly enrich our world state representations. Instead of trying to learn representations of blocks and structures purely from the limited number of Minecraft dialogues, or by having humans annotate our structures, we investigate how we can learn better world representations by pretraining richer models on auxiliary tasks using synthetic block configurations.

Chapter 5: Representing the World: Shape Localization

In the previous chapter, we manually added linguistic structure to our data by defining a set of Architect dialogue acts and using them to annotate utterances in the Minecraft Dialogue Corpus. The resulting annotations enabled us to explore the auxiliary tasks of Architect dialogue act classification and prediction; these can not only provide additional meaningful signals for downstream tasks that are more involved (such as utterance generation), but also afford us more control over and interpretability of our end-to-end models’ outputs. But this approach is not without its drawbacks: adding a second round of human annotation is expensive and noisy, and still limits us to using the narrow range of dialogues within the corpus itself.

Since the structures in the Minecraft world are amenable to synthetic data generation approaches, as generating variants only requires us to populate discrete 3D grids, we now turn our attention to ways we can enrich our world representations using automatic methods. As we have seen in Chapter 2, these structures are often composed of meaningful concepts (such as rows, towers, diagonals, planes, etc.) that are frequently referred to by humans, suggesting that the ability to explicitly locate and ground such concepts in the grid is necessary for both Architects and Builders alike. To this end, we design a model that does just this by training it to specifically focus on the task of locating these concepts in synthetically generated data that features these concepts in abundance.

In this chapter, we explore the task of identifying substructures in Minecraft target structures by pretraining a model that can identify and locate these concepts, or *shapes*, in the Minecraft world. This work is built upon that of Lambert et al. (2019) [83] who generated a corpus of synthetic 3D grids and used it to pretrain models that can perform shape *recognition* (i.e., identify the existence of various shapes) in the world. In Section 5.1, we describe their original synthetic data generation framework, as well as how we extend this framework to support the harder task of shape *localization*, defined in Section 5.2. We describe our models that use convolutional neural networks (CNNs) for this task in Section 5.3, and present and analyze shape localization results of this model on the synthetic data in Section 5.4. We show the results of applying this pretrained model to the structures in the Minecraft Dialogue Corpus in Section 5.5. Finally, we describe initial efforts to build a CNN-based Architect model in Section 5.6.

5.1 SYNTHETIC 3D BLOCK CONFIGURATIONS

In this section, we provide an overview of the synthetic grid data generation framework, originally designed for a multilabel shape recognition task, adapted from Lambert et al. (2019) [83]. Below, we describe how the framework defines a number of elementary shapes, then randomly samples them to generate 3D grids containing combinations of those shapes.

Elementary shapes Originally, Lambert et al. (2019) defined ten unique 3D shapes: rows, diagonals, T-shapes, L-shapes, U-shapes, planes, squares, rings, rectangular prisms, and cubes. Because rings, rectangular prisms, and cubes are fairly rare in actual Minecraft structures, we do not use these shapes to generate the corpus, instead choosing to focus on the remaining seven. For any given type, a shape’s construction is deterministic given its size, color, orientation, and starting block position. Furthermore, a shape’s label is defined using both its name and its orientation (e.g. `row_Y` for vertical towers, vs. `row_X` or `row_Z` for horizontal rows along the xz plane) in order to distinguish between similar shapes of varying orientations. The elementary shapes are defined as follows:

- **Rows** can range from 2 to 11 blocks in length, and can point in three different directions (`X`, `Y` and `Z`), aligning to each of the x , y , and z planes.
- **Diagonals** consist of at least 2 blocks, and can either be planar (along any of the xy , yz , or xz planes) or 3D (cross diagonals). Diagonals are categorized into three different types based on their orientation: `horizontal` diagonals lie along the xz plane, `vertical` along either xy or yz , and `cross` span all three dimensions.
- **T-shapes** are at least 3×3 in size and are composed of two orthogonal `rows`, in which one end of one row intersects the midpoint of another. The orientations of these rows determine the orientation of the T. `horizontal` T-shapes lie completely along the xz plane, while `vertical` ones span either xy or yz . Additionally, `vertical` T-shapes can be `up` (facing right-side up) or `down` (upside-down).
- **L-shapes**, which are at least 2×2 in size, are similar to T-shapes in construction in that they consist of two orthogonal `rows` attached at their ends. The orientations of L-shapes are similar to those of T-shapes, including `horizontal` L-shapes as well as `vertical` ones that can point either `up` or `down`.
- **U-shapes** are symmetrical structures also built out of rows: one row (with a length of at least 3) constitutes its base, while two orthogonal rows that are parallel to each other (with equal lengths of at least 2) constitute the sides. A U is constructed by first

specifying an L subshape, then attaching another `row` to its end in the appropriate direction. U-shape orientations are also similar to those of T- and L-shapes, including `horizontal`, `vertical_up` and `vertical_down`.

- **Planes** are 2D shapes with a side length of at least 2. They can either be `horizontal` along the xz plane or `vertical` along the xy or yz planes.
- **Squares** are subsets of planes whose sides are of equal length. They are always also labeled as `planes` and share the same orientations.

Composite shapes In the original data, composite shapes are formed by combining 2 to 4 randomly sampled elementary shapes in a single 3D grid. Each shape has a randomly sampled color and is placed into the grid such that none of the individual shapes touch each other. To increase the data complexity to better match that of actual Minecraft target structures, we increase the maximum number of elementary shapes in a grid to 8 and allow for shapes of different colors to touch each other.

Noise In order to capture arbitrary blocks that do not belong to these predefined shapes, noise blocks with an explicit `nothing` label can be randomly generated and added to any grid. In the original data, up to 90 noise blocks are generated at random locations with a randomly selected color; the set of noise blocks is then filtered to contain only those blocks that do not touch or intersect with the preexisting shapes. After applying bug fixes to the original noise generation implementation, we extend this to allow for multiple colors of noise blocks that may touch other shapes of different colors.

Shape labels There are 20 shape labels overall, 19 for each of the shape types with their various orientations along the explicit `nothing` label for randomly-added noise:

$$L \in \{\text{nothing, row_X, row_Y, row_Z, d_cross, d_horizontal, d_vertical,} \\ \text{T_horizontal, T_vertical_down, T_vertical_up, L_horizontal,} \\ \text{L_vertical_down, L_vertical_up, U_horizontal, U_vertical_down,} \\ \text{U_vertical_up, plane_horizontal, plane_vertical, square_horizontal,} \\ \text{square_vertical}\}$$

Labels are assigned for each elementary shape within a grid, as well as their subshapes. For example, a `U_vertical_up` shape is also classified as the `L_vertical_up` and `row` shapes

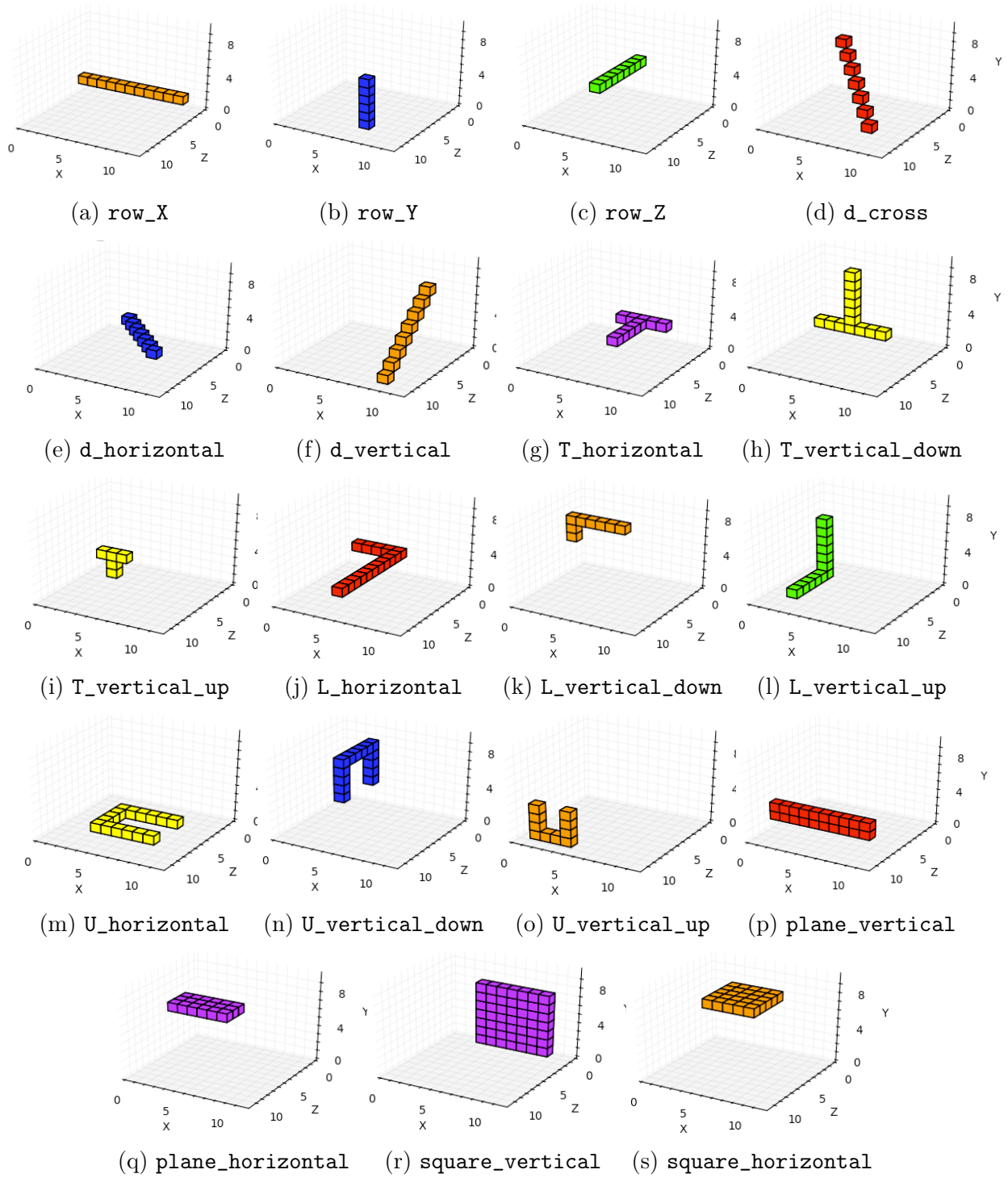


Figure 5.1: Elementary shapes and their associated labels.

that comprise it; all `square_horizontal` shapes are also labeled as `plane_horizontal`; and so on. Figure 5.1 shows a visualization of each type of shape.

Shape recognition Finally, Lambert et al. (2019) originally used this synthetic data to train CNN-based models for a multilabel shape recognition task, defined as follows: given an input 3D block configuration, the task is to classify it as containing one or more elementary shapes with their associated orientations. For this task, they designed a CNN model that recognized rows, diagonal lines, T-shapes, U-shapes, L-shapes, rectangles, and rectangular prisms (with their respective orientations, amounting to 18 labels overall) with near-perfect accuracy. They also reported minor improvements to Architect utterance generation performance by integrating the predictions from this shape recognizer into the original block counters model of Section 3.4, but we have since not been able to reproduce these improvements after transitioning to the updated block counters v2 model.

5.2 SHAPE LOCALIZATION

Shape recognition does not provide any information about *how many* of these shapes exist or *where* these shapes are located within the grid. This is especially pertinent for rows (also known as columns or towers, depending on the orientation), which are frequently used to build more involved shapes such as Ls and Us and commonly referred to as basic concepts in multi-step instructions and descriptions.

To this end, we define the task of shape localization: given an input 3D block configuration, the task is to label each block (or voxel) in the grid as belonging to one or more elementary shapes, as defined in Section 5.1. Shape localization is more challenging than shape recognition because, instead of a single prediction for an entire grid, labels must be assigned to every voxel within that grid such that the final prediction follows recognizable and meaningful shape boundaries. In this section, we provide a brief overview of how we adapt the synthetic shape generation framework to support the localization task and examples of the resulting generated corpus.

Coordinate-level shape labels To generate data for the shape localization task, we augment the existing data generation platform to also generate shape labels at the coordinate level for the synthetic 3D grids. For an input $11 \times 9 \times 11$ grid, we generate 20-dimensional vectors for each of the cells in that grid corresponding to the 20 overall shape labels (including an explicit `nothing` label used to indicate noise blocks that do not exist as part of any shape). In shapes that are composed from other elementary shapes (e.g. rows in L, T, and U shapes)

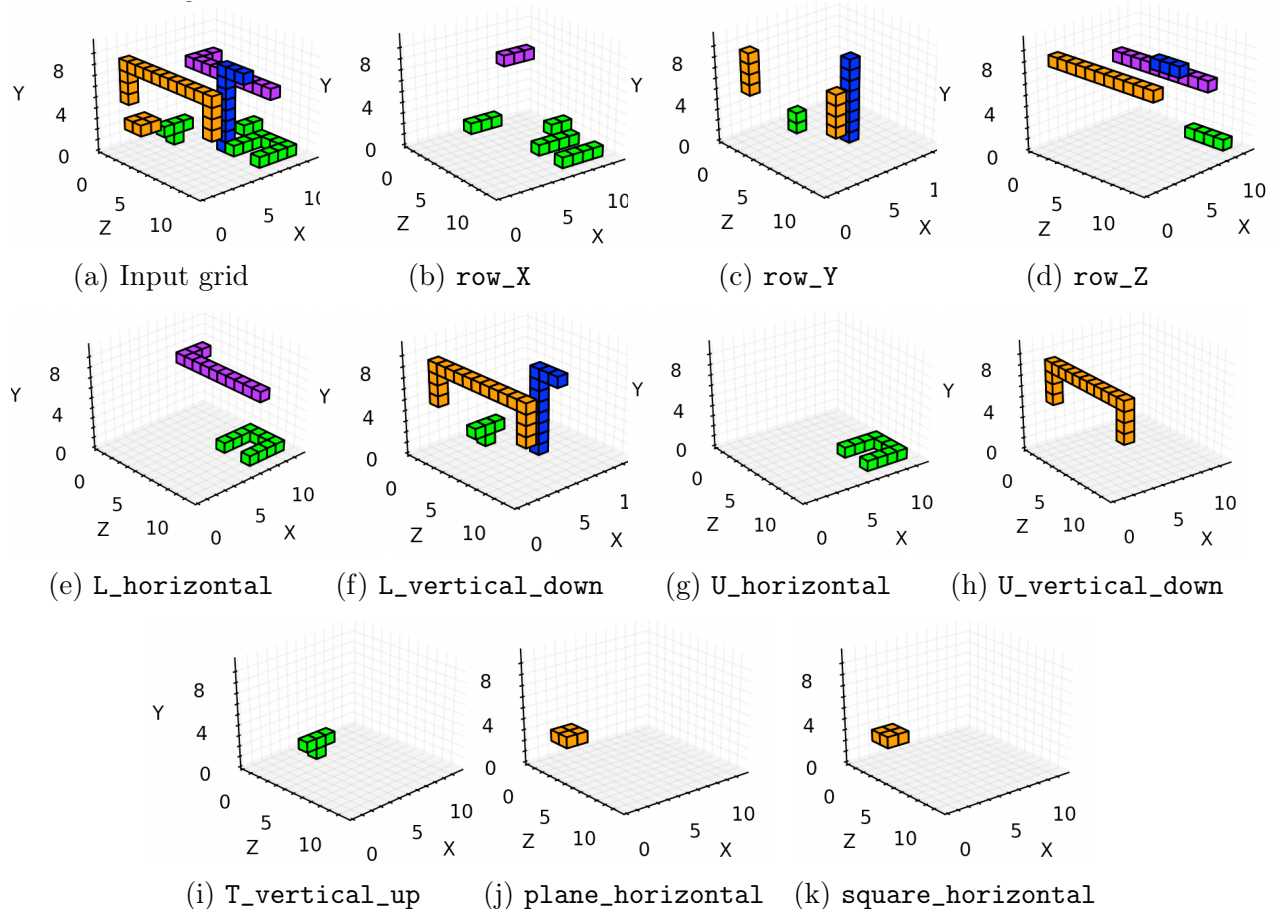


Figure 5.2: Example grid from the shape localization corpus. (a) represents the full grid. (b)–(d) show the row subshapes that the larger shapes are composed of. (e)–(i) show the L, T, and U shapes, while (j) and (k) show the labeled square (also labeled as a plane).

or are considered equivalent (e.g. planes as squares), corresponding cells in those shapes are assigned a multi-hot vector identifying the multiple classes of shapes they belong to. Empty cells are labeled with zero-vectors.

Corpus We generate a total of 100,000 synthetic shape grids (consisting of 90,000 composite shapes and 10,000 single shapes). Noise blocks are randomly added to 1/4 of the grids. We then randomly split the data into training, test, and dev sets of 64,000, 10,000 and 16,000 grids respectively. Figures 5.2 and 5.3 illustrate some examples of shapes and their localized coordinates in our data. Table 5.1 shows individual shape counts across the entire dataset, where the `nothing` count corresponds to the number of grids with randomly-generated noise.

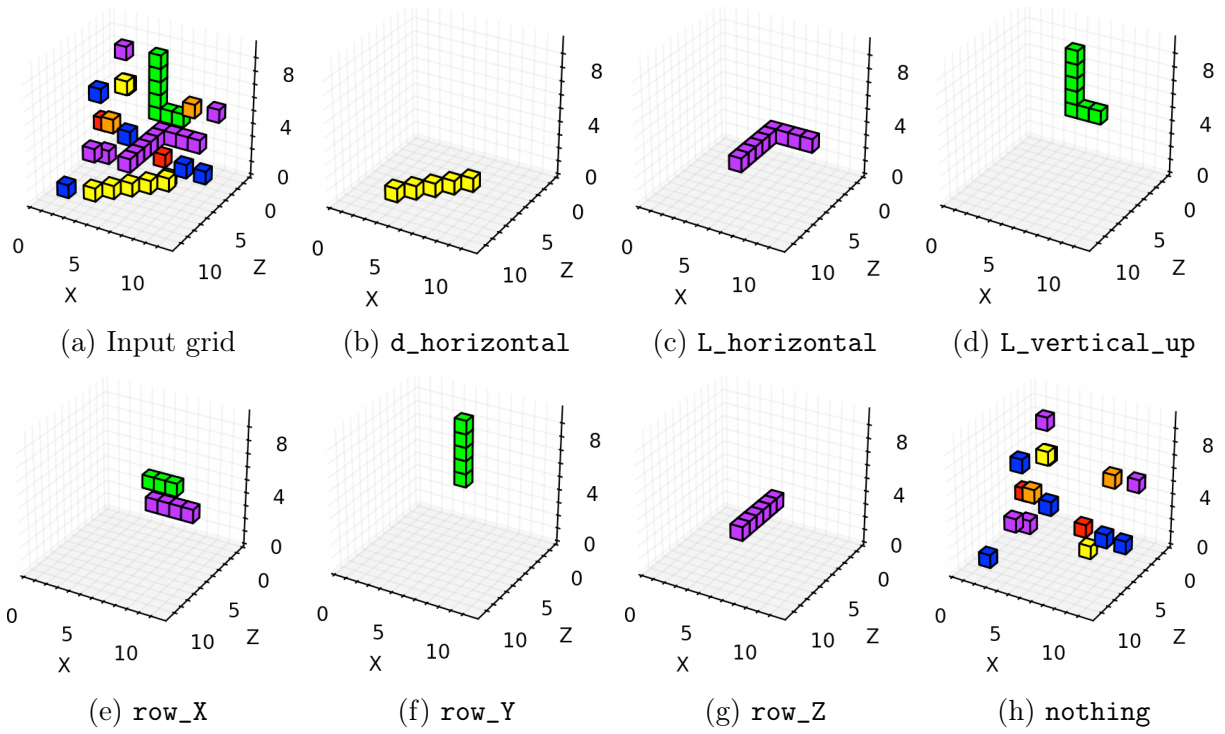


Figure 5.3: Example grid from the shape localization corpus with randomly-generated noise. (a) represents the full grid. (b) shows a horizontal diagonal, while (c)–(d) show the L-shapes. (e)–(g) decompose the L-shapes into their individual row labels. (h) shows the multicolored, randomly-generated noise blocks, labeled explicitly as **nothing**.

Shape	Count	Shape	Count
nothing	23101	d_cross	11882
L_horizontal	76082	d_horizontal	19311
L_vertical_down	60311	d_vertical	34490
L_vertical_up	60467	plane_horizontal	48606
T_horizontal	25204	plane_vertical	82591
T_vertical_down	20432	square_horizontal	23828
T_vertical_up	20233	square_vertical	42093
U_horizontal	25818	row_X	172850
U_vertical_down	19996	row_Y	179793
U_vertical_up	19949	row_Z	172808

Table 5.1: Counts of shape types in the overall synthetic shapes corpus.

5.3 CONVOLUTIONAL MODELS FOR SHAPE LOCALIZATION

Having generated a large synthetic corpus of grids annotated at the block level with shape information, we now describe our convolutional models for the shape localization task. Our model is based on the convolutional-deconvolutional network architecture commonly used in semantic segmentation tasks [84, 85]. We first provide some technical background on convolutional neural networks, then describe our model for shape localization in detail.

5.3.1 Technical Background

Convolutional neural networks (CNNs) [43, 86] are specialized neural network architectures designed to process grid-like inputs, such as 1D time series data, 2D grids of image pixels and 3D grids of voxels. Figure 5.4 shows a high-level overview of a CNN architecture. Typical convolutional blocks involve three main stages: convolution, nonlinear activation (or detector), and pooling.

Convolutions are specialized linear mathematical operations which are essentially weighted average operations across a given series. For example, a 1D discrete convolution $s(t) = (x \otimes w)(t)$ over a time series of outputs $x(t)$ is defined as:

$$s(t) = (x \otimes w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \quad (5.1)$$

where a is the age of a measurement and $w(a)$ is a weighting function.

Convolutions can also be computed over multiple axes at a time. For example, on a 2D input image I and using a 2D kernel K , the 2D convolution is calculated as:

$$S(i, j) = (I \otimes K)(i, j) = \sum_m \sum_n I(m, n)K(i-m, j-n) \quad (5.2)$$

Convolutions learn grid features by convolving over the grid with a small sliding window (called kernels or filters) with a size that is much smaller than the original input grid to produce a feature map. By sharing parameters over the entire input grid, kernels can learn to detect small, meaningful features such as

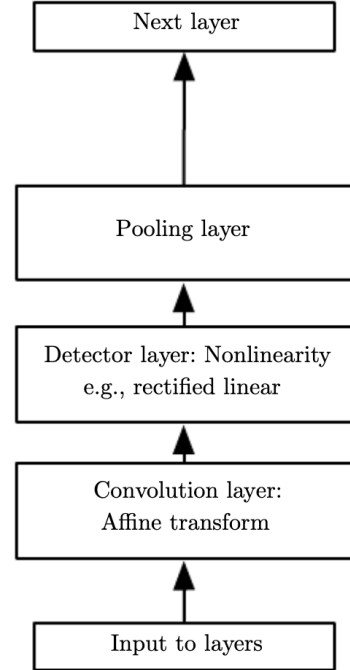


Figure 5.4: High-level overview of a convolutional neural network architecture [43].

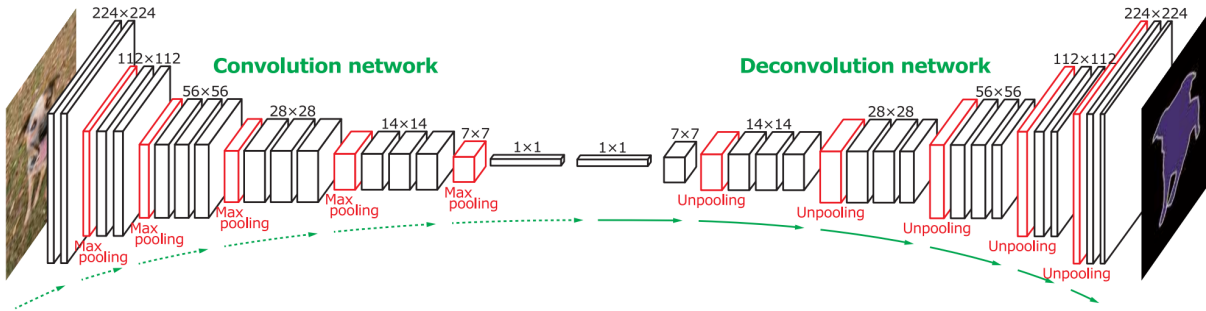


Figure 5.5: DeConvNet [84]: An example of a convolutional-deconvolutional network for semantic segmentation.

edges in an efficient manner.

After the convolution step, each feature map is run through a nonlinear activation function. A popular choice is the Rectified Linear Unit (ReLU) [87], an element-wise operation that replaces all negative values with zero:

$$f(x) = x^+ = \max(0, x) \quad (5.3)$$

In the final stage, a pooling function is used to replace the CNN output at a certain location in the grid with a summary statistic of nearby outputs. One common pooling strategy includes max pooling [88], in which the maximum output value within a rectangular neighborhood of the given location is used. Pooling is used to help make the representation more robust to small translations of the input.

Deep convolutional neural networks often involve a number of cascading blocks, each involving convolution, nonlinear activation, and pooling. CNNs act as feature extractors that transform input grids to multidimensional feature representations.

Deconvolutional networks are neural architectures that have successfully been used in conjunction with CNNs to perform object segmentation [84, 85]. Deconvolutional networks can be viewed as a mirrored version of a standard convolutional network that uses the CNN’s output feature representation to produce pixel-wise labels. In semantic segmentation models, e.g. DeConvNet [84] (Figure 5.5), a series of convolutional layers with max pooling extracts features from an input image by downsampling it to a compressed representation; that intermediate representation is then upsampled back to the original image size using a series of deconvolution (transposed convolution) layers with max unpooling, after which each pixel is labeled with a particular object class. The deconvolution and max unpooling layers in the deconvolutional network perform the inverse operation of the convolution and max

pooling layers in the convolutional network, respectively. As a result, the deconvolutional filters effectively learn to reconstruct the shape of the object in the input grid.

5.3.2 Shape Localization Model

We view shape localization as a multilabel voxel labeling task. To this end, our model is designed with a similar architecture to semantic segmentation models, using 3D convolution followed by deconvolution layers to process the block configurations. In the original DeConvNet object segmentation model, a pretrained 16-layer VGG net [89] was used as the convolutional network; we instead train the entire convolutional-deconvolutional network end-to-end. Additionally, we have found it useful to omit the max pooling and unpooling layers. The overall network architecture can be seen in Figure 5.6.

3D convolutional encoder The encoder consists of series of 4 3D convolution layers (kernel size of 3, stride of 1, and padding of 2), each followed by a ReLU activation function, that encode the input grid. Given that blocks in the grid can only be one of six block colors, we represent each cell state as a 6-dimensional one-hot vector, yielding a $11 \times 9 \times 11 \times 6$ world representation encoding the presence of blocks of a given color at any grid cell. The first convolutional layer contains 32 output channels; the number of channels then doubles at every layer, resulting in 256 output channels after the last convolutional layer. This results in an intermediate representation of the grid of size $19 \times 17 \times 19 \times 256$ which is fed as input to the 3D deconvolutional decoder.

3D deconvolutional decoder The decoder consists of a series of 4 3D deconvolution layers (kernel size of 3, stride of 1, and padding of 2), each followed by a ReLU activation function, that restore the compressed representation back to the original grid size. Mirroring the encoder, the number of output channels at every layer are halved;

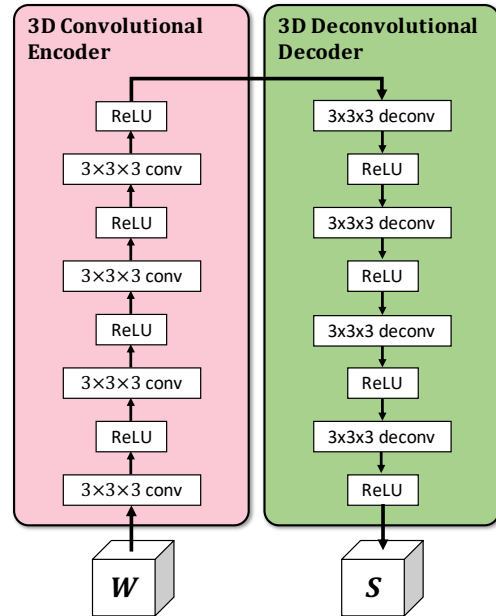


Figure 5.6: The shapes localizer CNN model. The $11 \times 9 \times 11 \times 6$ world state grid \mathbf{W} is encoded using 3D convolutions, then decoded using 3D deconvolutions to a grid of shape predictions \mathbf{S} of size $11 \times 9 \times 11 \times 20$.

the final layer has 20 output channels corresponding to the 20 shape labels that should be assigned to every block location in the original grid. We obtain a final output grid of size $11 \times 9 \times 11 \times 20$, and generate prediction probabilities for each of the 20 classes by applying a sigmoid over the output. We assign the final shape labels by thresholding this result with a value of 0.5.

5.4 RESULTS AND ANALYSIS

5.4.1 Experimental Setup

Data Our training, test, and development splits contain 64,000, 20,000, and 16,000 randomly sampled grids, respectively.

Training We trained for a maximum of 1000 epochs using SGD with a learning rate of 0.0001 (but our best models converged after no more than 51 epochs). We do not make use of dropout or batch normalization when training this model. During training, we minimize binary cross-entropy loss, summed across all classes, on the nonzero input voxels (i.e., we mask the model’s output such that only those corresponding to occupied block locations contribute to the loss). Similar to the multilabel dialogue act prediction model in Section 4.4, we make use of class weighting in the loss function to counteract the imbalance of class labels in the data (e.g., blocks in diagonal lines appear much less frequently than others). We stop training early when the micro-averaged F1 score on the held-out development set had decreased monotonically for ten epochs.

Prediction To assign shape labels to blocks in an input grid, for each voxel, we threshold the output probabilities for each of the shape classes with a value of 0.5. To help reduce spurious labels, we also apply the following constraints to every voxel in an output prediction:

- **Spurious shape labels.** A shape is defined to have 2 or more adjacent blocks of the same color (with the exception of diagonal lines, where adjacency is defined along the diagonals). For every block, we check that for every shape label, there exists an adjacent block of the same color with the same shape label. If not, we remove that particular shape label for the block. In effect, this removes spurious predicted shapes that consist of single blocks.
- **Spurious nothing labels.** `nothing` labels should not co-occur with other shape labels, and `nothing` blocks should not touch any other blocks of the same color (though

Shape	development				test			
	p	r	F1	support	p	r	F1	support
nothing	84.8	95.7	89.9	25574	84.1	96.0	89.7	30070
L_horizontal	98.2	98.7	98.4	129566	98.2	98.6	98.4	162065
L_vertical_down	98.4	99.0	98.7	95118	98.3	99.0	98.7	117992
L_vertical_up	97.6	99.2	98.4	93642	97.5	99.2	98.4	118576
T_horizontal	94.7	96.9	95.8	36132	94.9	97.1	96.0	44496
T_vertical_down	97.3	98.2	97.8	27110	97.2	98.3	97.7	33272
T_vertical_up	97.6	98.3	98.0	27265	97.4	98.2	97.8	33775
U_horizontal	93.9	93.2	93.6	59269	93.3	93.2	93.3	73581
U_vertical_down	94.3	97.5	95.9	41430	94.5	98.0	96.2	52442
U_vertical_up	94.7	98.0	96.3	40742	94.9	97.6	96.2	52882
d_cross	89.6	95.5	92.4	6130	90.6	96.0	93.2	7819
d_horizontal	97.0	99.0	98.0	11710	96.9	98.9	97.9	14534
d_vertical	95.5	98.7	97.1	19691	95.8	98.5	97.1	24821
plane_horizontal	99.9	100.0	100.0	143781	99.9	100.0	100.0	178001
plane_vertical	99.9	100.0	99.9	214041	100.0	100.0	100.0	268707
row_X	99.7	99.9	99.8	140772	99.8	99.9	99.8	175806
row_Y	99.7	99.9	99.8	127869	99.7	99.9	99.8	160210
row_Z	99.7	99.9	99.8	140007	99.7	99.9	99.8	175425
square_horizontal	97.5	99.5	98.5	59349	97.1	99.2	98.1	70974
square_vertical	97.0	99.6	98.3	91735	96.9	99.6	98.2	114291
Micro average	98.1	99.1	98.6	1529933	98.0	99.1	98.5	1909739

Table 5.2: Shape localization results without constrained decoding on the development and test sets.

this may still occur if no other shape predictions were made for an adjacent set of blocks). The **nothing** label is removed from a given block if it violates either of these conditions.

- **Empty predictions.** Finally, if a block exists in the grid but was not assigned a label due to an empty prediction, it is assigned the **nothing** label by default.

5.4.2 Results

In Table 5.2, we first evaluate the performance of each shape class individually without using constrained decoding, and report precision, recall, F1 scores and support for each class. We also report the micro-averaged precision, recall, F1 score and support across classes. One drawback of evaluating on the voxel level is that it does not evaluate how well the model can

Shape	development				test			
	p	r	F1	support	p	r	F1	support
nothing	94.7	95.8	95.2	24574	94.5	95.9	95.2	30070
L_horizontal	98.9	98.6	98.7	129566	98.9	98.4	98.7	162065
L_vertical_down	99.2	98.9	99.0	95118	99.2	99.0	99.1	117992
L_vertical_up	98.7	99.2	98.9	93642	98.7	99.1	98.9	118576
T_horizontal	97.3	96.6	96.9	36132	97.6	96.8	97.2	44496
T_vertical_down	99.1	98.0	98.5	27110	99.0	98.0	98.5	33272
T_vertical_up	99.3	98.1	98.7	27265	99.1	98.0	98.5	33775
U_horizontal	95.3	92.6	93.9	59269	94.7	92.6	93.7	73581
U_vertical_down	95.7	97.3	96.5	41430	95.9	97.8	96.8	52442
U_vertical_up	96.0	97.8	96.9	40742	96.1	97.4	96.8	52882
d_cross	95.9	93.7	94.8	6130	96.1	94.2	95.2	7819
d_horizontal	98.3	98.5	98.5	11710	98.3	98.3	98.3	14534
d_vertical	98.0	98.0	98.0	19691	98.1	97.8	97.9	24821
plane_horizontal	100.0	100.0	100.0	143781	100.0	100.0	100.0	178001
plane_vertical	100.0	100.0	100.0	214041	100.0	100.0	100.0	268707
row_X	99.9	99.8	99.8	140772	99.9	99.8	99.9	175806
row_Y	99.8	99.8	99.8	127869	99.8	99.9	99.9	160210
row_Z	99.9	99.8	99.9	140007	99.9	99.8	99.9	175425
square_horizontal	97.8	99.4	98.6	59349	97.4	99.2	98.3	70974
square_vertical	97.5	99.5	98.5	91735	97.4	99.5	98.5	114291
Micro average	98.9	99.0	98.9	1529933	98.8	99.0	98.9	1909739

Table 5.3: Shape localization results with constrained decoding on the development and test sets.

recover overall shape boundaries; another possible evaluation we can consider is to calculate accuracy scores on a shape basis, but we do not do this here.

Generally, the model performs strongly when evaluated on synthetic data. Performance on rows, which are prevalent in the data as subshape labels, and planes, which contain a large number of blocks, is extremely strong. The more complex L, T, and U shapes are slightly harder for the model. Though performance on cross diagonals suffers from the fewer number of examples in the corpus, our model is still able to achieve a reasonable level of performance on the sparser class with the use of class weighting during training. Unsurprisingly, the randomly-generated **nothing** noise blocks are hardest for our model to label correctly. We see that the model performs better on recall over precision, suggesting that our model tends to overgenerate labels.

In Table 5.3, we therefore investigate the effect of adding constrained decoding. By adding simple constraints to remove spurious labels, our model enjoys increased precision

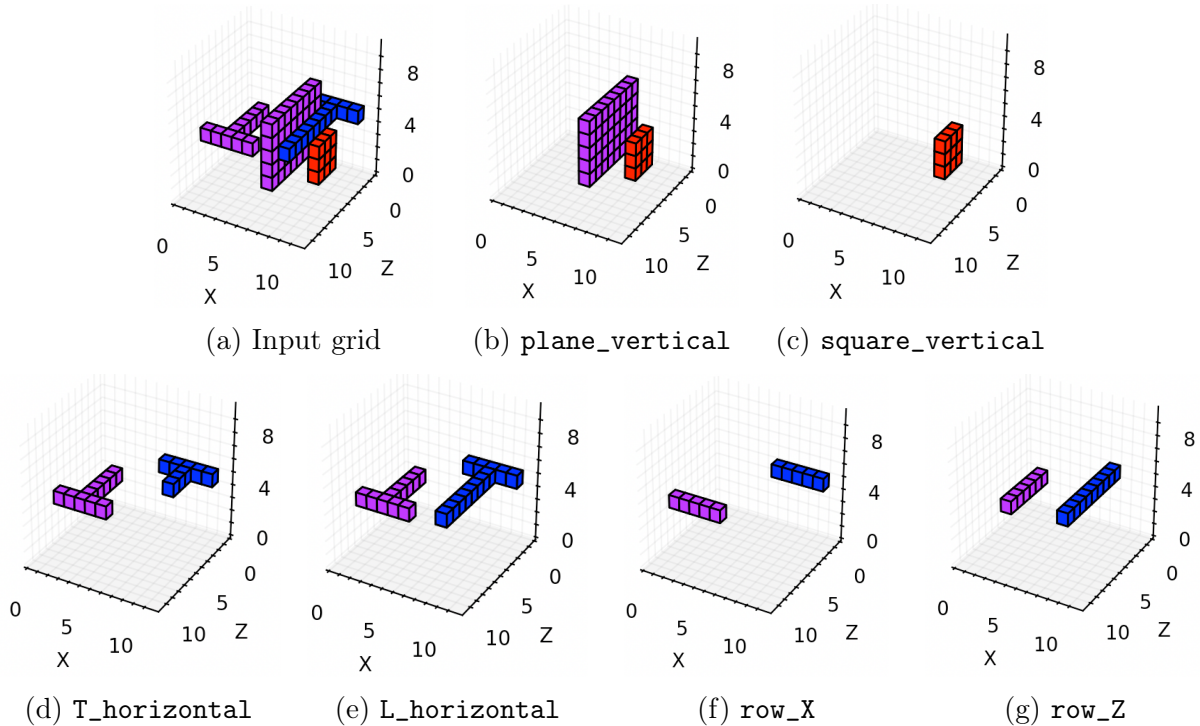


Figure 5.7: Example 5.1

across shapes without sacrificing performance on recall. The effect is especially apparent for **nothing** blocks, which show a near 10-point increase on precision in both development and test sets. Constrained decoding also boosts precision slightly on the more complex L, T, and U shapes and greatly for cross diagonals (6.3 point increased development set precision, with similar improvement on test).

5.4.3 Qualitative Analysis

Here, we show some examples of our model’s predictions on various development set block configurations.

Example 5.1. Figure 5.7 shows our model’s predictions on a fairly simple block configuration consisting of two clearly-defined T-shapes, a square, and a plane. The model is able to recognize both the purple and red planes while also correctly identifying that only the red plane should also be labeled as a square. The model also decomposes the horizontal T-shapes into their respective L-shapes and rows. This example also showcases the model’s ability to produce multilabel outputs: the intersection blocks in the purple and blue T-shapes both embody 4 shape labels (T_horizontal, L_horizontal, row_X, and row_Z).

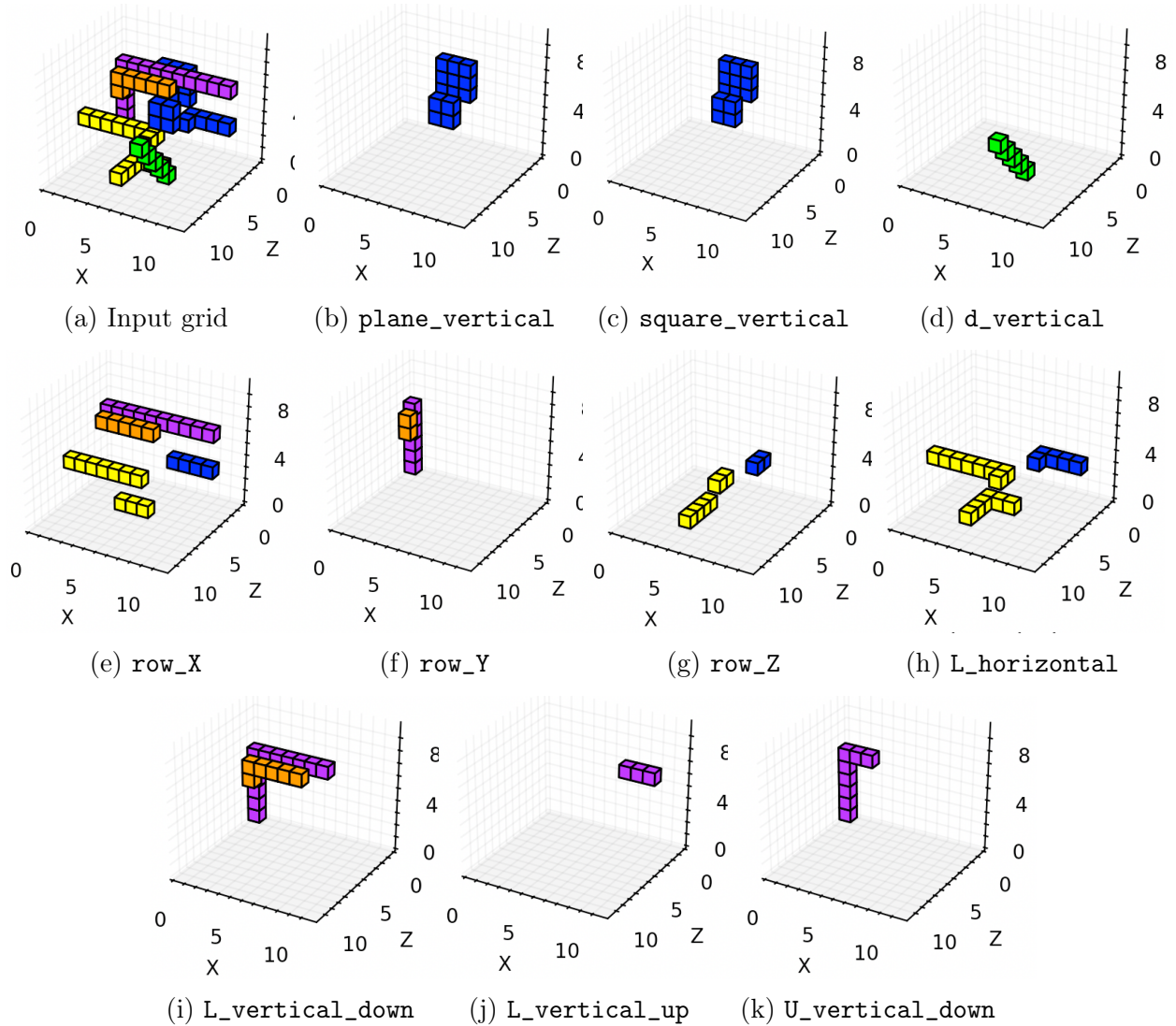


Figure 5.8: Example 5.2

Example 5.2. Figure 5.8 shows our model’s predictions on a more complicated block configuration that includes L-shapes of various orientations, squares, and a diagonal line. The model’s predictions are mostly correct, as it is able to identify individual rows, the diagonal line, squares, and horizontal L-shapes. However, it does not classify the purple downwards vertical L-shape in (i) fully and overgenerates, instead labeling parts of its stem as an upwards vertical L-shape in (j) and the rest a downwards vertical U-shape in (k).

Example 5.3. Figure 5.9 shows our model’s predictions on a simple block configuration consisting of a horizontal U-shape and randomly generated noise. The model is able to distinguish the red U-shape from the noise, which itself contains a red noise block.

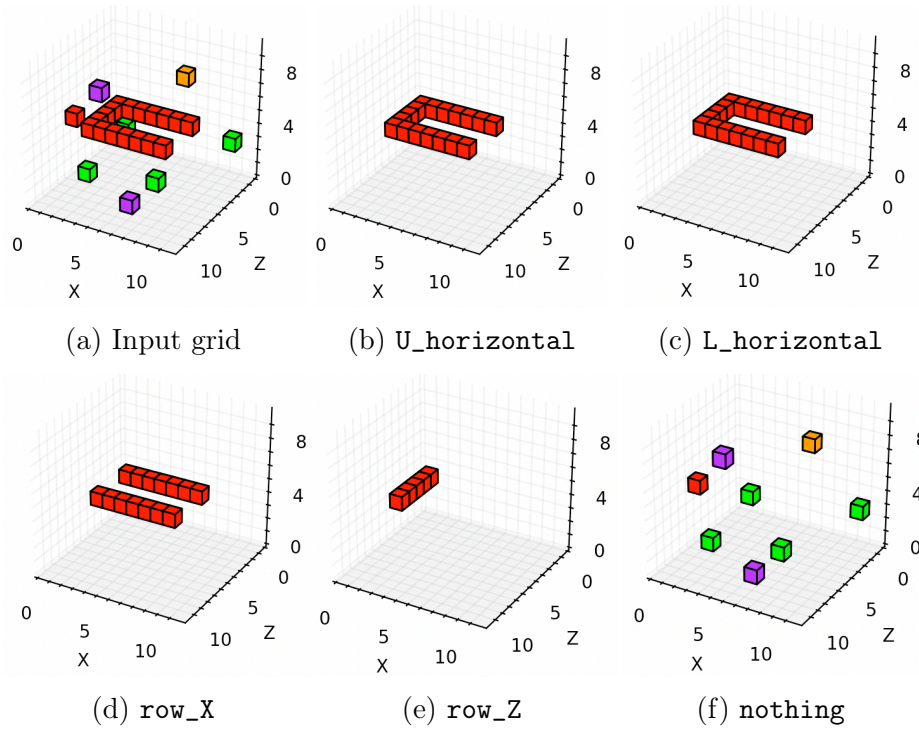


Figure 5.9: Example 5.3

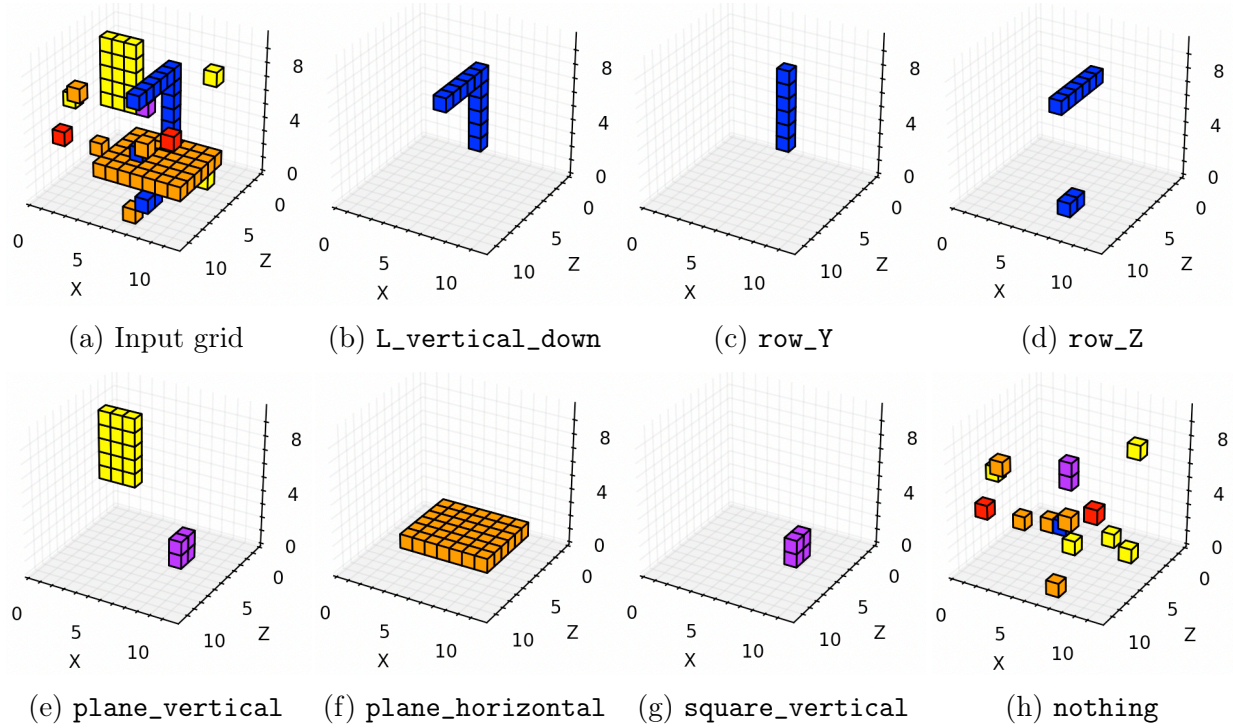


Figure 5.10: Example 5.4

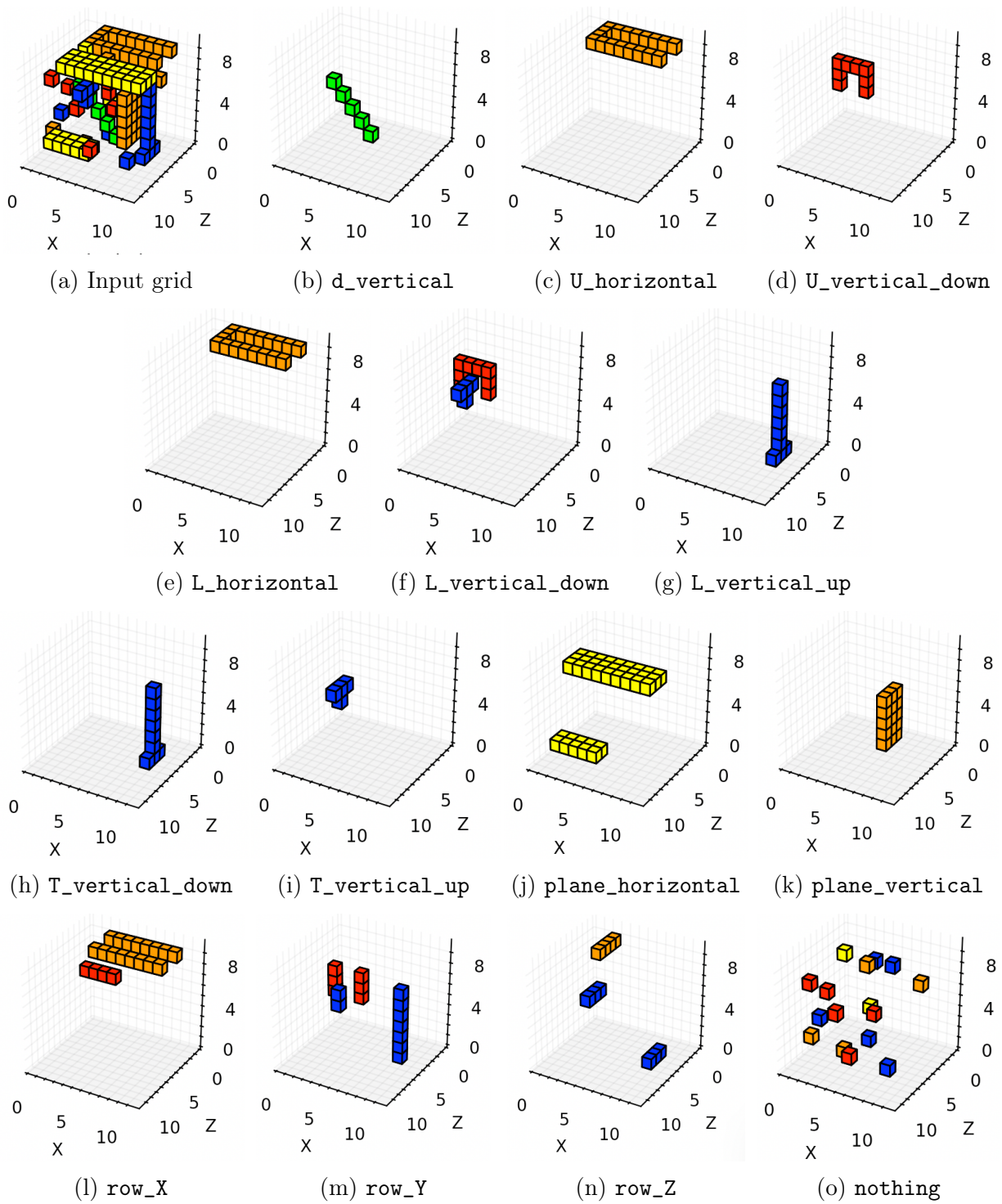


Figure 5.11: Example 5.5

Example 5.4. Figure 5.10 involves a more complex block configuration involving multiple shapes and multicolored noise. The model is able to identify almost all the shapes, but overlooks a short vertical purple row of 2 blocks, mistakenly labeling these as **nothing**.

Example 5.5. Figure 5.11 features an even more complex block configuration containing 14 distinct shape labels, including the explicit **nothing** label. Despite the added complexity of the structure, the model is able to decompose it into elementary shapes without fail.

5.5 LOCALIZING SHAPES IN MINECRAFT TARGET STRUCTURES

In the previous section, we showed that, barring some issues of overgenerating multilabel predictions that can be somewhat alleviated with constrained decoding, our shape localizer model could identify reasonable shape boundaries in synthetically generated block configurations. However, these synthetic configurations are still relatively simple when compared to the actual target structures in our Minecraft data (for examples, we refer back to Figure 2.3, which visualizes some Minecraft target structures). Minecraft structures are much more densely populated with blocks and feature elementary shapes that are often located closely together instead of scattered within the grid. But, since our main goal is to build better world representations for actual Minecraft structures, we now analyze how well our model, trained on a large amount of synthetic data, can locate elementary shapes in real Minecraft target structures.

Example 5.6. Figure 5.12 shows our model’s predictions on a target structure that resembles a red chair sitting at a table with a purple tabletop and two blue legs. The model identifies reasonable boundaries for both horizontal and vertical planes in (b) and (c), but produces incomplete predictions for vertical squares in (d). One reason for this could be

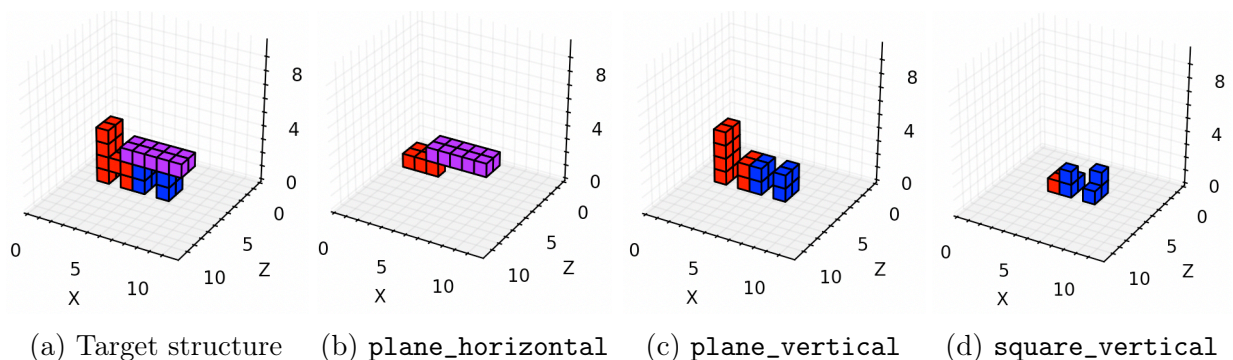


Figure 5.12: Example 5.6

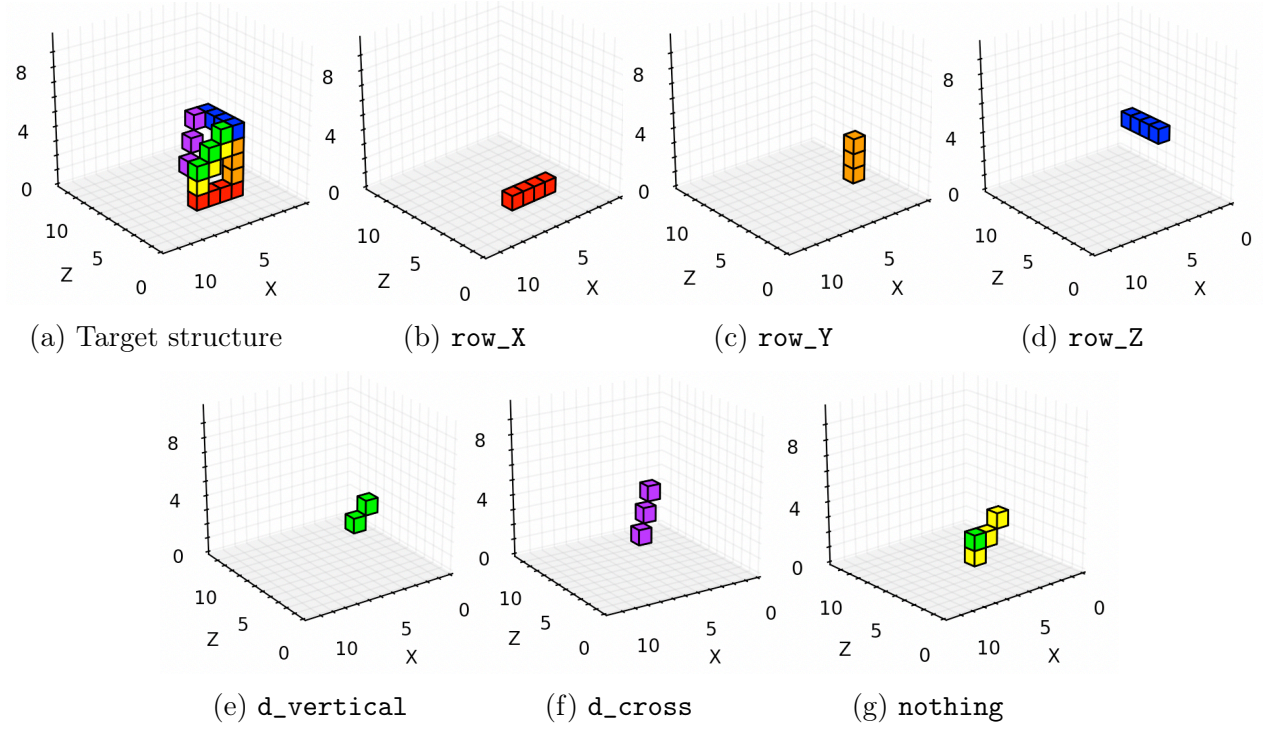


Figure 5.13: Example 5.7

because the synthetic data generation does not explicitly try to produce meaningful intersections between planes in its structures, which is a somewhat common phenomenon in real Minecraft target structures.

Example 5.7. Figure 5.13 shows our model’s predictions on a target structure that forms multiple triangular shapes using diagonal lines. The model is able to identify isolated rows in (b)–(d) and diagonal lines in (e) and (f), but labels the yellow diagonal and part of the

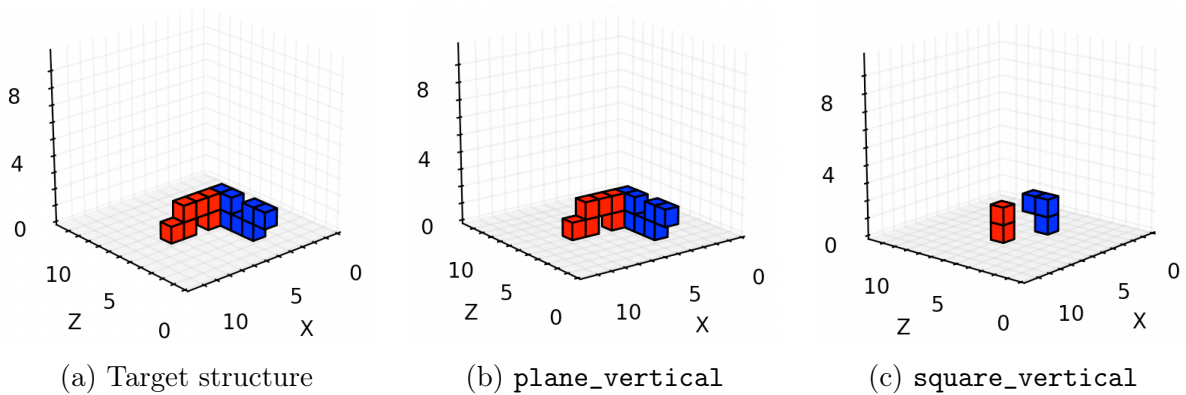


Figure 5.14: Example 5.8

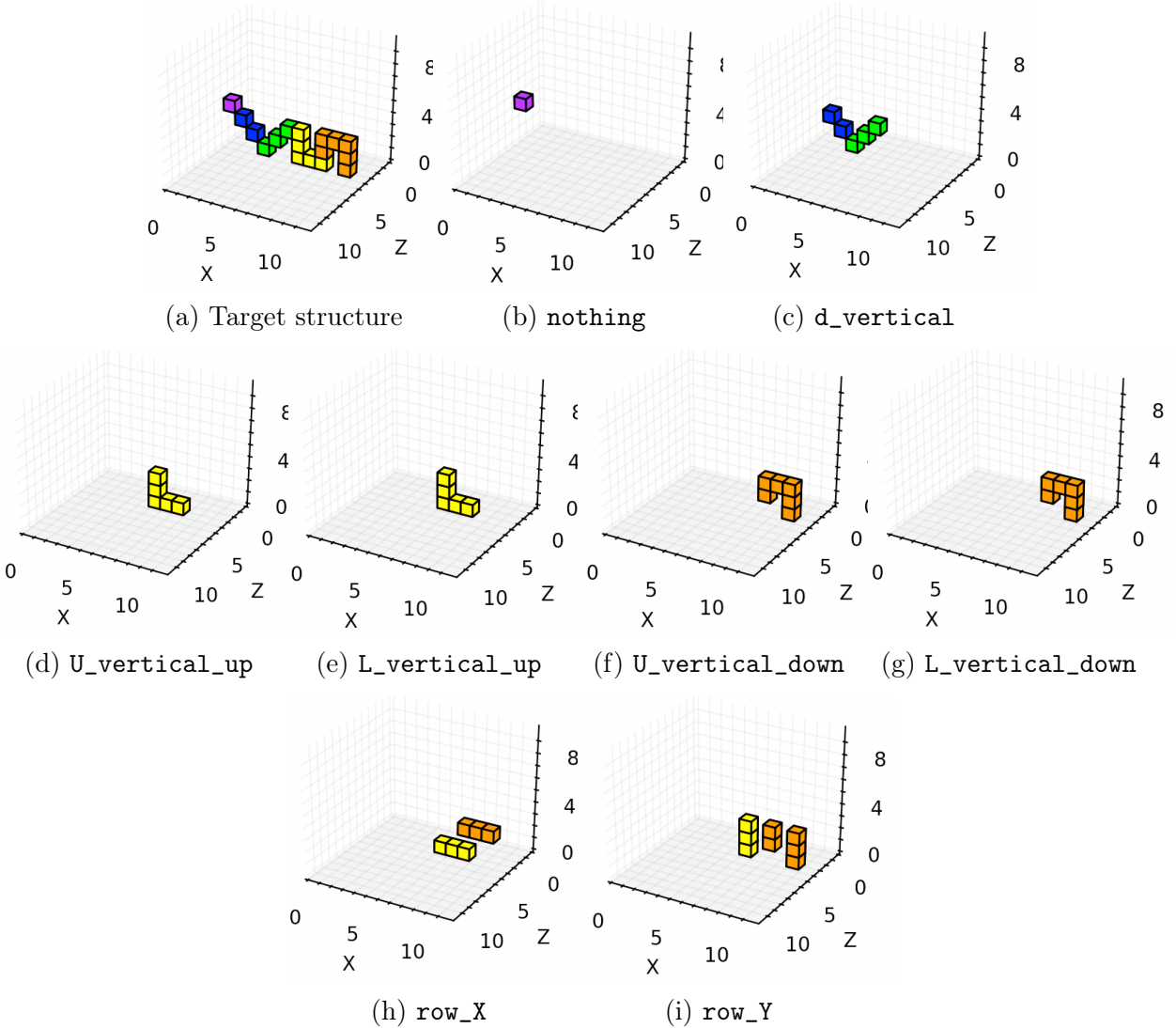


Figure 5.15: Example 5.9

green diagonal as **nothing** in (g) instead of as another vertical diagonal. This likely occurs because perfectly juxtaposed shapes are extremely unlikely in the synthetic data.

Example 5.8. Figure 5.14 shows our model’s predictions on a target structure consisting of interlocking red and blue substructures. Unfortunately, the model’s predictions here are questionable: though the substructures themselves are planar, the predicted shapes do not actually resemble planes or squares. At the same time, it is somewhat unclear how the substructures should be annotated using our current shape definitions.

Example 5.9. Figure 5.15 shows our model’s predictions on a planar structure of a zig-zagging line. The model can clearly identify portions of the line by color, including diagonal

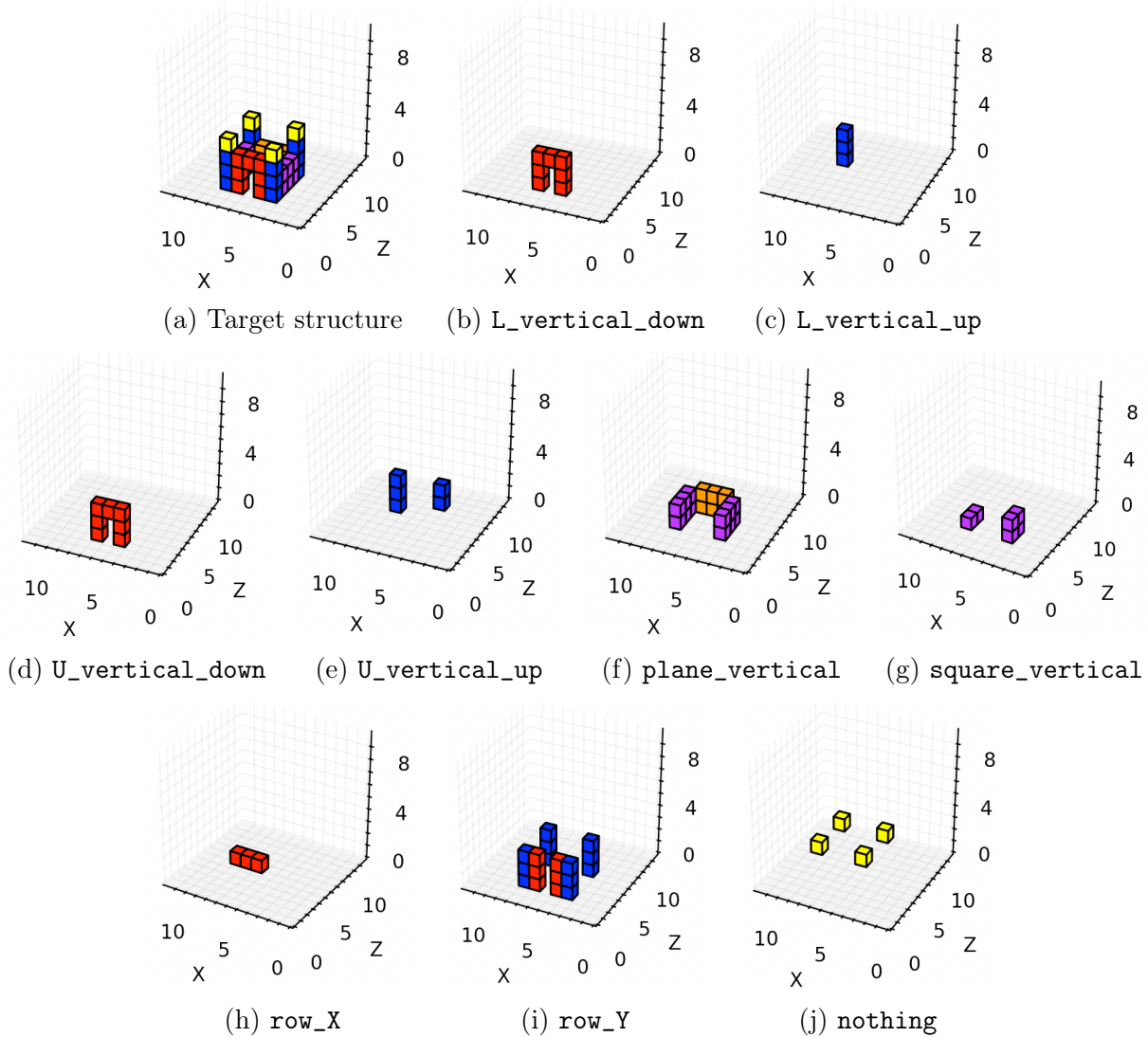


Figure 5.16: Example 5.10

lines in (c) and L-shapes in (e) and (g), but also predicts incorrect and oblong U-shapes in (d) and (f), respectively.

Example 5.10. Figure 5.16 shows our model’s predictions on a 3D structure resembling a castle. The model is able to identify some elements, such as the red downwards U-shape in (d), purple and orange planar walls in (f), blue vertical pillars in (i) and yellow miscellaneous blocks in (j). However, the model also produces some unrecognizable shape boundaries labeled as upwards vertical U- and L-shapes and squares in (e), (c), and (g), respectively.

Qualitative analysis of our model’s predictions on Minecraft target structures shows that the model has learned a representation of elementary shapes from synthetic block configurations well enough to somewhat generalize to real Minecraft structures. In particular, our

model more reliably identifies simpler elementary shapes (such as rows and diagonals), and it localizes shapes especially well if they are of a distinct color and spatially isolated from other shapes. However, the model still exhibits issues of label overgeneration, struggles to fully identify more complex shapes (such as T-shapes, U-shapes and squares) and generates meaningless predictions for target structures that may not be composed of recognizable elementary shapes (such as in Example 5.8). Some of these issues may be mitigated by increasing the complexity of the synthetic data to include more meaningful intersections between shape boundaries, while others may require different approaches to model training and inference.

5.6 TOWARDS A CNN-BASED ARCHITECT

In a similar fashion to including Architect dialogue acts in Chapter 4, we would like to include the shape predictions on Minecraft target structures as additional structured input for Architect utterance generation models. As described above, the output of the shapes localizer is a grid of multi-label predictions, with one vector of predictions per cell in the $11 \times 9 \times 11$ input grid. One straightforward way we could integrate the output grid of predictions into an utterance generation model is to use another CNN-based module responsible for encoding the world state to process them as additional input channels. Jayannavar et al. (2020) [38] have found recent success on the alternate task of Builder Action Prediction (BAP) by building CNN-based world state encoders into an end-to-end sequence-to-sequence action predictor. In this section, we briefly describe our recent efforts to build a CNN-based Architect utterance model, inspired by the BAP architecture, that we could use as a baseline to which we can add shape information.

5.6.1 CNN World State Encoder

As before, the Architect must consider how the target structure can be constructed successfully given the current state of the built structure. In our previous block counters model, to capture this, we considered all possible optimal translational and rotational alignment variants between *Built* and *Target* in order to summarize their difference. Here, we describe how we represent the two structures as input to a CNN world state encoder, before we explain how the world state encoder produces an embedding of that representation.

Choosing a single optimal alignment In the BAP model, the world state is the current grid configuration that is fed into the action prediction model at each time step. In the

Architect utterance generation task, however, we additionally have access to the (unaligned) target structure, and both *Built* and *Target* should be represented as relevant world state inputs. In the block counters model (Section 3.4), we estimate this by averaging counts of blocks to be placed and removed over all possible optimal alignments at a given point in the game.

While we could represent the same amount of uncertainty across all optimal alignments as input to a CNN encoder, we instead select a *single optimal alignment* to represent. In order to choose an optimal alignment that is consistent with the human Architect’s mental model during the game, we compute the optimal alignments that (1) exist between the completed *Built* and *Target* structures at the end of the game, and (2) persist throughout the duration of the game. Of that set, we choose one optimal alignment as our gold standard, and denote this as our *single optimal alignment*, $A^{*(1)}$. We compute one such single optimal alignment for each game. Intuitively, we are choosing an alignment between *Built* and *Target* that is likely to be one that the human Architect used to guide their instructions during the game.

Recall that a structure S ’s position and orientation can be mutated by an alignment A in which S undergoes a translation A_T (shift) followed by a rotation A_R , denoted $A(S) = A_R(A_T(S))$. Thus, once we have the single optimal alignment $A^{*(1)}$, we apply the necessary rotations and translations to *Built* in order to permute its blocks into *Target* space according to that alignment. As a result, we no longer have a summary representation of the differences between *Built* and *Target* across all possible alignments, but rather an explicit representation of blocks in both structures under a single optimal alignment.

Representing the raw world state Minecraft blocks are unit cubes that can be placed at integer-valued $\langle x, y, z \rangle$ locations in a 3D grid; the Collaborative Building Task restricts these to a build region of size $11 \times 9 \times 11$. By aligning *Built* and *Target* according to the single optimal alignment as described above, we can represent each of the aligned *Built* and *Target* structures as two sets of input channels to our world state CNN encoder. For a given structure, we represent each cell state as a 7-dim one-hot vector encoding the presence (or absence) of blocks of one of the 6 colors, or empty, at any grid cell. Concatenating these representations for the two structures, our model represents each grid cell as a 14-dim one-hot vector, yielding a $11 \times 9 \times 11 \times 14$ minimal world state representation, denoted W .

CNN world state encoder Our world state encoder architecture is inspired by the world state encoder architecture of Jayannavar et al. (2020). To obtain a representation of each grid cell, we feed the raw world state tensor W through a multi-layer CNN that embeds each grid cell conditioned on its neighborhood. The model consists of k 3d-conv layers with kernel

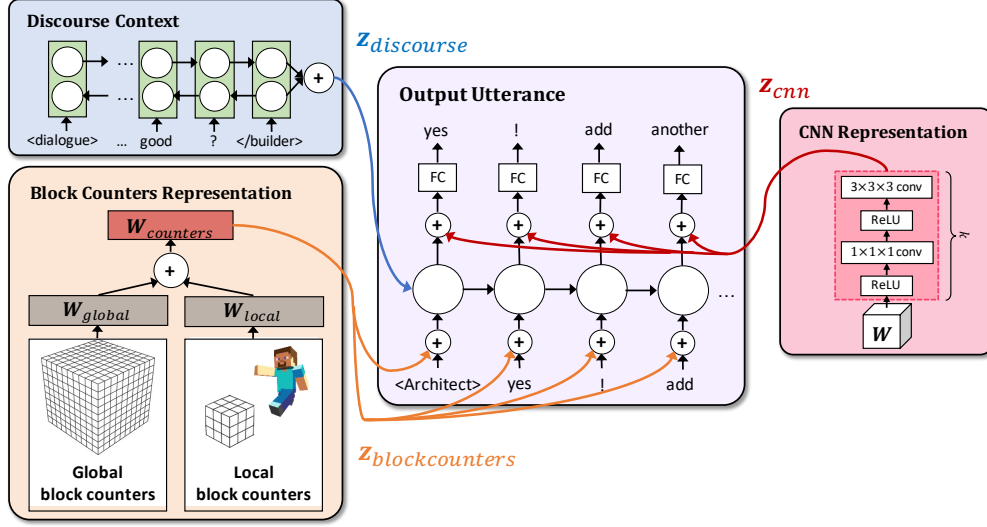


Figure 5.17: The block counters v2 model with an added CNN world state encoder. The CNN encoder produces a global representation of *Built* and *Target*.

size 3 (CNN₃), stride 1 and no padding, followed by a ReLU activation function. Between every successive pair of these layers is a $1 \times 1 \times 1$ 3d-conv layer (CNN₁) with stride 1 and no padding, for dimensionality reduction purposes, again followed by ReLU. We denote the final representation produced by our CNN encoder as z_{CNN} . A key difference between our model architecture and that of Jayannavar et al. (2020) is the use of padding in the CNN₃: while the BAP model keeps the grid size consistent by padding the input, we intentionally reduce the size of the grid at every layer in order for the output representation to retain a manageable dimensionality for the output utterance decoder.

Adding z_{CNN} to block counters v2 Our final Architect utterance model, pictured in Figure 5.17, integrates the final CNN world state representation z_{CNN} as input to the output utterance decoder while retaining the original block counters encoding at the decoder’s input. Specifically, we add z_{CNN} to the decoder’s output at every time step and feed the result through a network of fully connected layers, the output of which is then fed to a final linear layer to generate the next token.

5.6.2 Results and Discussion

Experimental setup We use the same data splits as those used by the block counters v2 model. After data cleaning, our training, test and dev splits contain 6,422, 2,855, and 2,251 samples. We perform a grid search over CNN hyperparameters, but report the model

Model	BLEU				Precision / Recall				spatial	dialogue
	B-1	B-2	B-3	B-4	all keywords	colors				
counters v2	18.0	9.2	5.3	3.3	15.7 / 15.3	21.2 / 28.5			11.1 / 9.8	19.0 / 22.3
CNN Architect	16.0	7.3	3.8	2.2	11.8 / 10.8	7.9 / 7.5			7.7 / 6.7	21.2 / 14.7

Table 5.4: BLEU and term-specific precision and recall scores on the development set.

that achieves the highest BLEU score, which involves a 4-layer world state encoder CNN, whose output is combined with the decoder’s output and further embedded using a 3-layer feedforward network, alongside a 1-layer bidirectional RNN dialogue history encoder. For these experiments, we do not make use of constrained decoding in order to investigate the model’s raw ability to produce output with coherent colors.

Results We compare our CNN Architect model against the block counters v2 baseline (Section 3.7) in Table 5.4. Unfortunately, while our newer model features a more complex world state representation that has been shown to be useful for the BAP task, its performance on the Architect utterance generation task is poorer across the board compared to the relatively simpler block counters v2 model. Most worrisome is the model’s now abysmal performance on both color precision and recall, suggesting that the representation of relevant colors has become too muddled, despite the model also making use of the original block counters encoder. As we have seen in Sections 3.8 and 4.6, we enjoy far better gains by adding better-informed inputs to simpler models, rather than attempting to learn complex representations completely end-to-end, as we do here. While we could attempt to add shape predictions to this baseline CNN model to observe its effects, we assert that this baseline performance is far too poor to be useful, and argue that it would be more sensible to first design a better baseline CNN-based Architect.

There are several possible reasons behind this negative result. First, involving a CNN adds to the model complexity, and results in a sizable increase in the number of trainable parameters. While this has been less of a problem for BAP (Jayannavar et al. (2020) showed that it is possible to train these richer representations with larger amounts of synthetic data), the training data and training objective for the Architect utterance generation task are vastly different: output Architect utterances consist of a large number of tokens, of which only some are noisily informative for a world state representation, while the output action sequences in BAP can be represented directly in a grid and are much more closely aligned with the input world state. Furthermore, because we are forced to reduce the output world state representation z_{CNN} to a manageable dimensionality, we lose valuable information about

the world in the process. Possible avenues for future work should explore different ways of training these richer representations specifically for the utterance generation task, either by pretraining them on alternative subtasks or using alternative loss functions, as well as better ways to obtain a concise but informative representation of the world without as much information loss, possibly by using some form of attention over the input grid.

5.7 CONCLUSION

In this chapter, we explored how to build better representations of the world by pretraining convolutional networks on the auxiliary task of shape localization. After generating a large amount of synthetic block configurations labeled with elementary shapes at the coordinate level, we built a convolutional-deconvolutional network that exhibited strong performance on unseen synthetic configurations. We then showed that despite being trained only on synthetic data, this model is able to localize the same types of elementary shapes in real Minecraft target structures to some degree, suggesting that it has learned a representation of the Minecraft world that is informative enough for grounding some simple higher-level concepts. However, its performance on real data is far from perfect, and future work should investigate how to improve these types of models further to provide even cleaner representations of these concepts in Minecraft structures.

Since describing our baseline Architect utterance model in Chapter 3, our focus has consistently become narrower, from modeling and leveraging the specific linguistic phenomena of Architect dialogue acts in Chapter 4 to learning to ground elementary concepts in 3D block configurations in this chapter. We now take a step back in order to regain focus on our original overall goal: to build interactive agents. In the next chapter, we discuss how we have applied some of the work in this thesis to actual collaborative scenarios by building fully interactive Architect agents.

Chapter 6: Designing an Interactive Architect

Throughout this dissertation, we have continually narrowed our focus from full-fledged interactive scenarios in the full Collaborative Building Task to smaller and more specific subtasks. While this has enabled easier automated evaluation of our systems, it does not give us a reliable idea of how well our systems would work in practice. Importantly, the end goal for a fully interactive Architect system is to successfully build full target structures by communicating with a human Builder. While our current metrics and utterance-level human evaluation can estimate a measure of task success by examining term-specific performance and utterance correctness, they do so by evaluating our systems in static contexts of specific human-human games. But in true interactive scenarios, these contexts can vary wildly, and these utterance-level metrics are not designed to capture how well our systems might do in human-machine scenarios with the likely possibility of cascading errors. Instead, it is necessary to evaluate our system’s performance at the dialogue level when interacting with actual humans in real-time games.

In the Architect utterance generation task, one of the simplifying assumptions we leverage is that we only consider points in a game where we know **A** must speak. This removes the requirement of needing to predict *when* it is appropriate to speak. Therefore, our first step to building a fully interactive Architect involves designing a simple dialogue manager to invoke our block counters Architect utterance model at appropriate moments in the game. Furthermore, given previous analysis of our model performance on specific types of utterances (in particular, descriptions and corrections), we implement rule-based quality-of-life improvements for the interactive Architect in an attempt to avoid known weaknesses.

In this chapter, we describe our efforts to design an interactive framework around our block counters v2 utterance model of Section 3.7, and perform a case-by-case analysis of its performance in various interactive scenarios.

6.1 DESIGNING AN INTERACTIVE ARCHITECT

Our first order of business is to enable our Architect utterance model (specifically, the block counters v2 model with constrained decoding of Section 3.7) to operate in real-time interactive games. The design of our dialogue manager is inspired by our previous work in building a rule-based dialogue management system in Narayan-Chen et al. (2017) [90], in which we design a dialogue mediator for a modular agent architecture, COG, for executing instructions in a 2D Blocks World domain.

COG combines a problem solving (planning) component with a basic language understanding and generation system in order to interact in a two-dimensional Blocks World domain where a human, who wants one or more shapes to be constructed on a grid, needs to communicate the goal of this planning task to the agent. In order to unify the components in COG, we design a dialogue mediator, or DM, that is responsible for guiding the interaction with the human and delegating the tasks of parsing, planning, and realization to various individual components. The DM is a finite state machine that accepts user-described goal descriptions and prompts the user to reword their utterances, clarify missing information, and define new shapes as needed. The DM is also responsible for keeping track of the cumulative information gained about a goal configuration over a dialogue sequence in order to backtrack the states of the semantic parsing and problem solving components if mistakes occur during the interaction.

In order to build an interactive Architect that can collaborate with human users on the Collaborative Building Task, we similarly need to design a dialogue manager that can invoke our underlying Architect utterance model at appropriate moments. However, since we use a single end-to-end system for utterance generation, our dialogue manager does not need to carefully define how information should be transferred between various internal components, though this also means that we cannot invoke well-defined clarification questions and explicit shape learning as we do in instruction parsing with COG. Despite these differences, the COG system still provides a good example of the capabilities needed for a robust dialogue manager, including the ability to accumulate and backtrack world states, allow for seamless error correction and inject variation in generated outputs as needed.

During an interactive Minecraft game, our rule-based dialogue manager accumulates a series of world state observations captured at triggering moments (when **B** places or removes a block, or when either player sends a chat message). When the dialogue manager determines that **A** should emit an utterance, it passes the entire sequence of historical world states to our model, which then processes them as if in a static context to generate the next **A** utterance. Below, we describe the design of the dialogue manager we use for our interactive Architect.

Knowing when to speak Instead of learning a policy for when to speak, we find that using a rule-based system to trigger the Architect model works well in practice. Given our model’s propensity to generate single-block instructions, we allow **A** to speak only one utterance at a time before waiting for **B** to react. If the utterance produced by **A** is too short (e.g. *“great!”* or *“perfect!”*), we force the model to generate a follow-up utterance. After every **B** action, we wait for 5 seconds before generating a new utterance, in case **B** continues to build blocks. After a **B** utterance, however, we assume that **B** requires an **A** response

and start generating immediately. If **A** is in the process of generating an utterance when it is interrupted by another **B** action, we restart generation using the updated world state.

Generating the first instruction Extensive qualitative analysis has shown that our model is particularly weak at generating appropriate instructions at the beginning of a game. In our corpus, the first utterances usually depict the target structure at a high level and provide information about where it should be built within the grid in order to respect the boundaries of the build region. However, the types of descriptions our model produces are extremely generic and usually incorrect, and our model tends to use *<unk>* tokens to describe rarely-seen target structures, e.g. *“we’re going to build a <unk>”* or *“we’re going to build a 3 x 3 square.”* Furthermore, the model on its own is not able to generate spatial relations with respect to the absolute grid, e.g. *“start at the bottom left corner.”*

Therefore, instead of invoking our model at the beginning of a game, we delegate the task of first utterance generation to a rule-based system. To do this, we first construct a heatmap of possible block locations along the ground by counting blocks of each color in each grid cell across all possible alignments between the empty build region and the target structure. Then, using that heatmap, we construct a templated first utterance as follows:

- If there is a block that can be placed at the center of the board, we choose the most likely color for that block using the heatmap and generate an utterance referring to it, e.g. *“place a red block in the center of the board.”*
- Otherwise, if a block can be placed near the center, but not exactly in the center, we choose the most likely color for one of the near-center blocks and generate an appropriate utterance, e.g. *“put an orange block down near the middle.”*
- If a block can be placed in one of the corners of the grid, we choose the most likely color for one of the corner blocks and generate an utterance, e.g. *“start with a purple block in the corner of the grid.”*
- Finally, if a block cannot be placed in the exact corner but in a location adjacent to it, we choose the most likely color for one of these locations, then generate, e.g. *“put a blue block down near a corner of the grid, but not exactly in the corner.”*

These rules can be applied to almost all target structures in our dataset to produce a valid first utterance. However, these rules fail if a structure floats completely off the ground. Regardless, we do not consider structures that have floating elements, as our model is unable to generate the language needed to express how to build floating blocks in the first place.

Backtracking from errors Another area our model struggles with is in recognizing mistakes and generating appropriate corrections. The ability to recover from errors is crucial to completing a game successfully, as we have seen in so many of the human-human game logs in Chapter 2. At the same time, we have seen in Chapter 4 that even when our model is conditioned on ground truth dialogue act information, in which explicit corrections are labeled, corrections are still extremely difficult for our model to generate correctly.

We therefore implement a strict rule-based approach to error correction in our dialogue management framework. Throughout a game, we monitor the Hamming distance between the target and built structures, T and S , with respect to all optimal alignments:

$$|\text{diff}_{A \in A^*}(T, S, A)| = |B_p \cup B_r| \quad (6.1)$$

where $A^* = \arg \min_A (|\text{diff}(T, S, A)|)$. That is, we monitor the total number of blocks to be placed and removed across all optimal alignments of the two structures.

When **B** places or removes a block, we compute all optimal alignments for the new built structure with respect to the target and calculate their Hamming distance with respect to these alignments. If the Hamming distance increases (i.e., a larger number of block placements and/or removals are required to transform *Built* to *Target*), we assume a mistake has been made and initiate a strict undo and backtracking mechanism as follows:

1. The dialogue manager interrupts gameplay and emits an undo utterance: *“sorry, please undo that action,”* then waits for **B** to undo their last action.
2. The history of world states is reset to the point in the game just before the mistake was made and used as input to the Architect utterance model.
3. The model generates a new utterance, which we filter to ensure it is different from the original **A** utterance that caused the error.
4. We consider the error resolved if the next **B** action does not increase the Hamming distance. Otherwise, the error state persists and we return to step 1.

We therefore offload the responsibility of generating appropriate corrections from the model and leave no room for mistakes to be made using this rule-based approach. However, it should be noted that this approach is not robust to placeholder blocks that are required for floating structures, as these will always temporarily increase the Hamming distance between *Built* and *Target* when placed. Additionally, this approach is not robust to errors in which multiple erroneous **B** actions have occurred, as it assumes a block-by-block execution of instructions.

Avoiding repetitiveness When the undo function is triggered, the Architect model is invoked again using a backtracked history of world states. However, because our model does not produce very diverse n -best outputs, it is still likely to generate an utterance that is extremely similar to the original one that caused the error. For example, if **A** first generates an erroneous “*put a blue block to the right of that,*” it is likely to continue to generate simple paraphrases of this incorrect utterance in later turns, e.g. by simply replacing the token *put* with *place* or replacing *of that* with *of it*.

In order to avoid multiple frustrating rounds of undo cycles in which **A** continually repeats itself, we therefore implement a more aggressive filtering mechanism to avoid repetitiveness in the model’s outputs. During every backtracking cycle in which an error has not yet been resolved, we compile a list of previous error utterances based on those generated by **A** against which we filter every new utterance generated by **A**. We compile this list using simple utterance paraphrases that we define with the following phrase equivalences:

- *put, place*
- *put a, place a, put one, place one*
- *of the one you just placed, of the one you just put down, of that one, of that, of it*

For example, if “*put a blue block to the right of that*” was an erroneous **A** utterance that was generated, we expand the list of paraphrased error utterances to also include “*place a blue block right of that,*” “*put one blue block right of that,*” “*put a blue block right of the one you just placed,*” “*place a blue block right of that one,*” and so on. This list is continuously extended with erroneous **A** utterances and their paraphrases, and is used to filter repetitive newly-generated **A** utterances, until the error has been resolved.

Ending the game Another weakness of our model is that it tends to continue issuing instructions even when *Built* fully matches *Target*. Instead of relying on the model to generate an appropriate final utterance, we force an end-of-game utterance, “*great, we are done!*” when the Hamming distance between *Built* and *Target* reaches zero.

Other features Finally, we implement some additional quality-of-life improvements. First, if too many retries have occurred in an error state, we further constrain model outputs by requiring that they mention colors, in order to avoid repeated generic utterances such as “*put another one on top of that.*” Furthermore, we prevent the model from generating utterances that include patterns which are generally incorrectly used (e.g. *a x b* or *2d drawing* phrases), tokens signifying the end of game (e.g. *done* or *we’re good*), and *<unk>* tokens in the output.

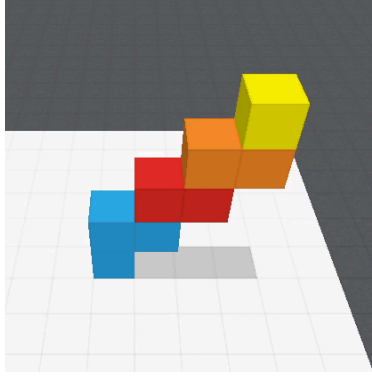


Figure 6.1: Target structure for Example 6.1.

6.2 INTERACTIVE ARCHITECT EXAMPLES

In this section, we walk through a couple of examples to illustrate our model’s performance in true interactive scenarios. In each of these examples, a human Builder **B** interacts with the block counters v2 Architect model **A** in real time.

Example 6.1. This example shows a game in which the target in Figure 6.1 is being built. The target structure is a simpler version of a similar target structure in the validation set.

Figure 6.2 At the beginning of the game, the rule-based first utterance generator begins the interaction by describing the location of the first block:

A: place a blue block in the center of the board

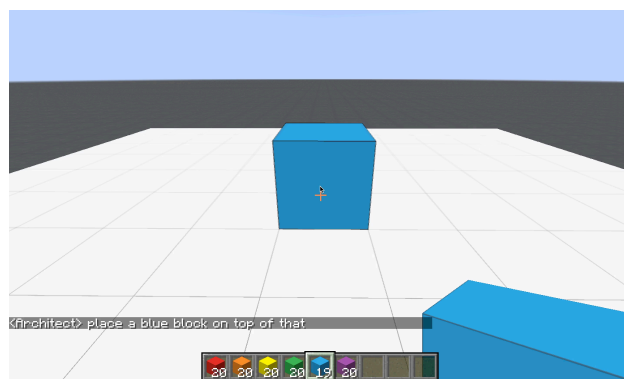


Figure 6.2

Figure 6.3 After the first utterance is generated deterministically, the model refers to the most recently placed block to issue a new instruction for (a). For the latter instruction (b),

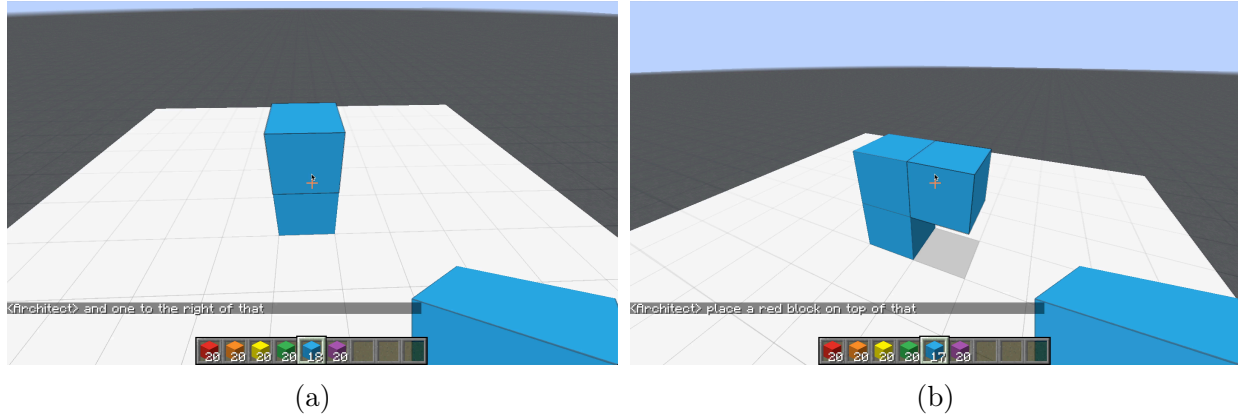


Figure 6.3

the model implicitly refers to another *blue block* with an appropriate spatial relation, *to the right of that*:

A: *place a blue block on top of that*

A: *and one to the right of that*

Figure 6.4 After finishing the blue subsection, the model appropriately switches to red blocks in (a), then again implicitly refers to another *red block* in (b) to continue the structure:

A: *place a **red** block on top of that*

A: *and one to the right of that*

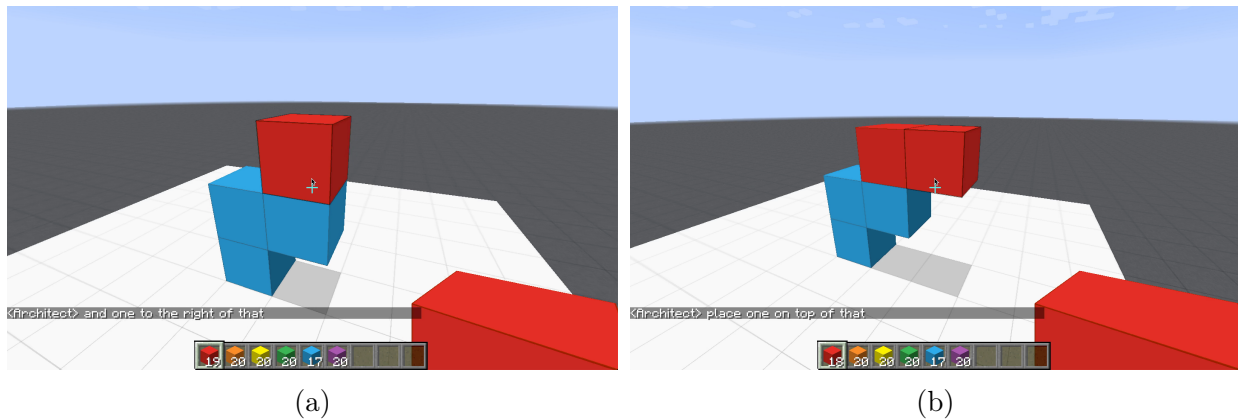


Figure 6.4

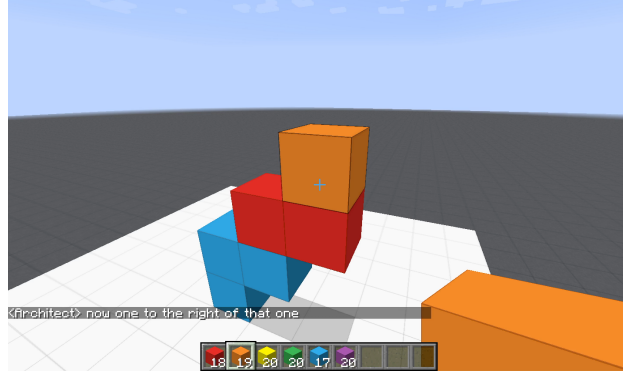


Figure 6.5

Figure 6.5 After **B** has placed the rightmost red block, **A** provides an underspecified utterance, intending to describe the orange block that should be placed on top of it. While **B** could have assumed the color of this new block to also be red, they instead ask a clarification question about the block’s color instead. While the model does not directly respond to **B**’s question, **A**’s next instruction is more color-specific than the previous instruction:

A: place one on top of that

B: what color?

*A: place one **orange** on top of that*

Figure 6.6 **A** continues to describe the orange subsection without explicitly referring to *orange blocks*:

A: now one to the right of that one



Figure 6.6

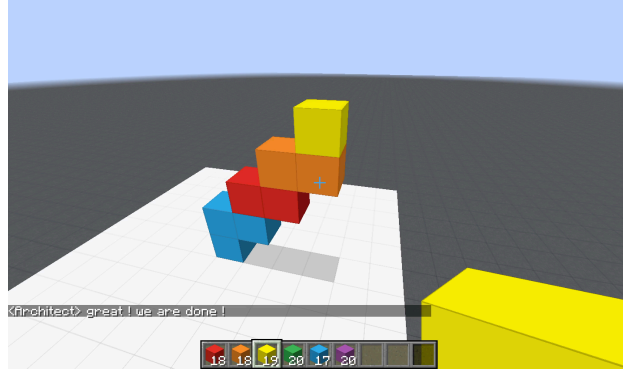


Figure 6.7

Figure 6.7 **A** provides another underspecified utterance to describe the last block. After **B** asks for further clarification about the block’s color, **A** replies with another indirect, but more specific, response:

A: *one on top of that*

B: *same color?*

A: *place a **yellow** block on top of the one you just placed*

Finally, after **B** places the last yellow block, the dialogue manager recognizes that *Built* now perfectly matches *Target* and issues an end-of-game utterance:

A: *great! we are done!*

This example showcases a simple structure that our interactive Architect is able to describe to completion. The model changes colors appropriately and generates the correct spatial relations in order to incrementally build the structure. Additionally, the model frequently

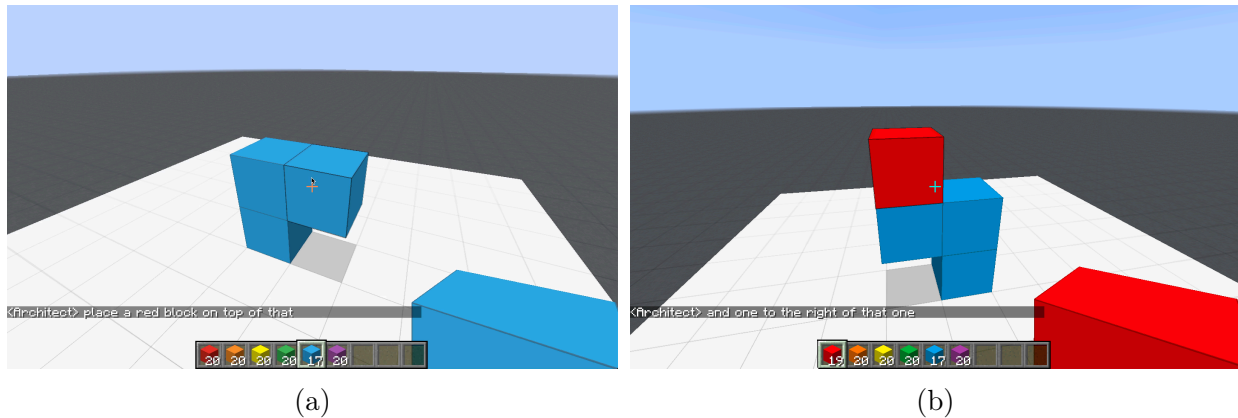


Figure 6.8

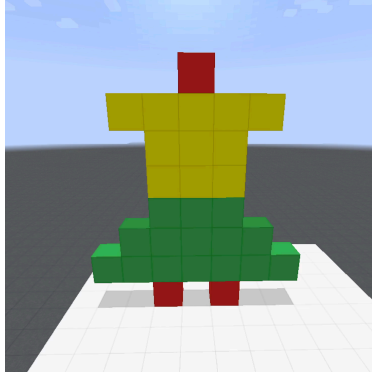


Figure 6.9: Target structure for Example 6.2.

refers to previous blocks in order to infer block colors. While the model does not directly answer questions when prompted, it is still able to provide some additional clarification.

However, this example can be misleading. We consider a similar scenario in Figure 6.8 for the same target structure, where **B** first turns to face *Built* from the other side before placing the first red block in (b). Here, we would expect our model to now use the spatial relation *to the left* to describe the next red block to be placed, but it instead continues as before, incorrectly generating *and one to the right of that one*. This suggests that our model is not actually robust to dynamic changes in **B**'s perspective – an important factor for success in real Minecraft dialogues – and that we may have lucked out by building a target structure that zig-zags upwards and to the right.

Example 6.2. This example shows a game in which the target in Figure 6.9 is being built. This structure is more complex than that of Example 6.1, involving more blocks constructed in a recognizable shape, but remains completely planar and involves no floating blocks.

Figure 6.10 At the beginning of the game, the rule-based first utterance generator starts the interaction, describing the location of one of the two red legs. After placing the first red block, **B** decides to take initiative in the conversation and asks for a description of the target structure, which **A** subsequently ignores, firing off instructions to place green blocks:

A: *place a red block in the center of the board*

B: *what are we building today?*

A: *place one green on top of that*

Figure 6.11 After continuing the column of green blocks, **B** asks yet another question, hoping to speed up the building process by eliciting more information about the next few

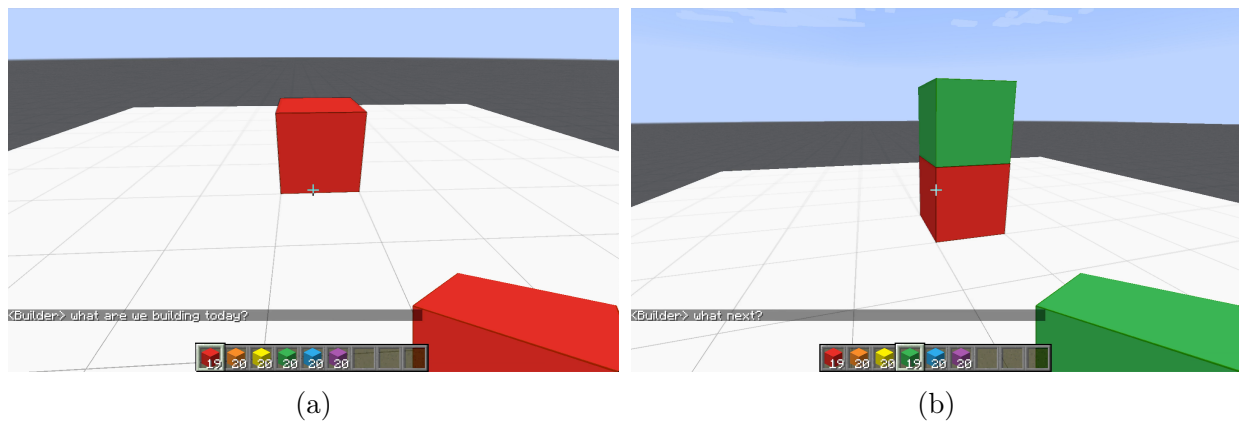


Figure 6.10

steps from **A**. However, **A** continues to ignore **B** and begins a familiar zig-zagging pattern of green blocks. **B** rotates around to the backside of the structure between actions in a change of perspective, but **A** is oblivious and continues on with further green block instructions:

B: *what next?*

A: *place a green block on top of the one you just placed*

B: *how many greens?*

A: *place one green to the right of the one you just placed*

Figure 6.12 In continuing the pattern of green blocks, **A** generates its first incorrect utterance, and placing another green block on top of the built structure (as in Figure 6.12b) violates *Target*. Here, the automatic undo function kicks in and immediately corrects **B**:

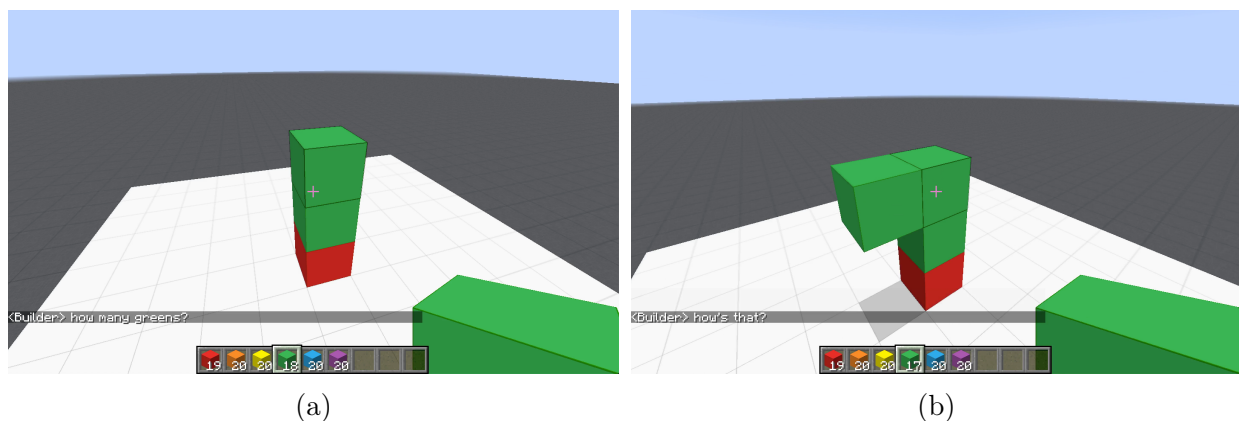


Figure 6.11

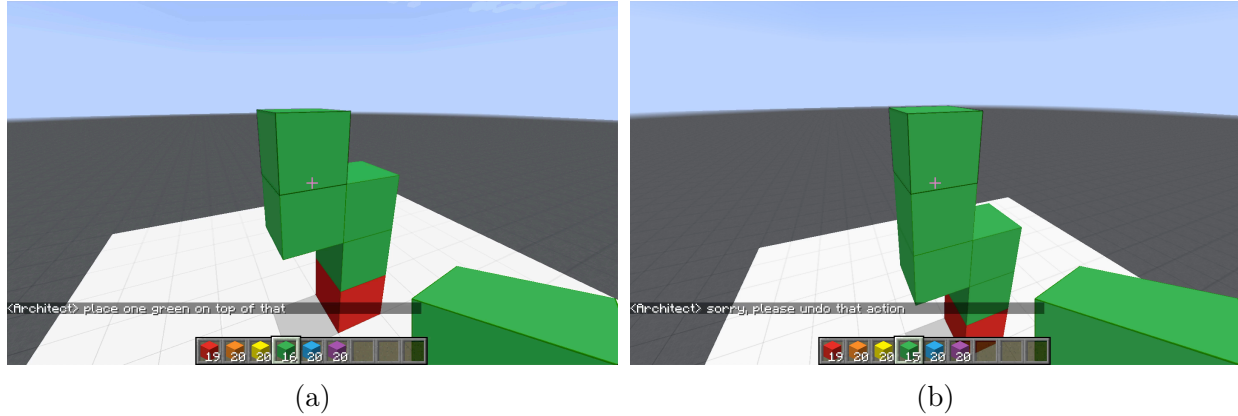


Figure 6.12

B: *how's that?*

A: *place a green block on top of that one*

A: *place one green on top of that*

A: *sorry, please undo that action*

Figure 6.13 After the game history has been rewind following **A**'s error, the model tries again to generate an appropriate instruction. However, as **B** points out, **A**'s new instruction is still far too similar to its previous incorrect instruction. While our dialogue manager tries to avoid this scenario by performing rule-based paraphrase matching against previous erroneous utterances, the matching criteria is clearly still too brittle, allowing a redundant failure to slip through. After some trial and error, **A** is finally able to generate a new utterance that is appropriate and correct:

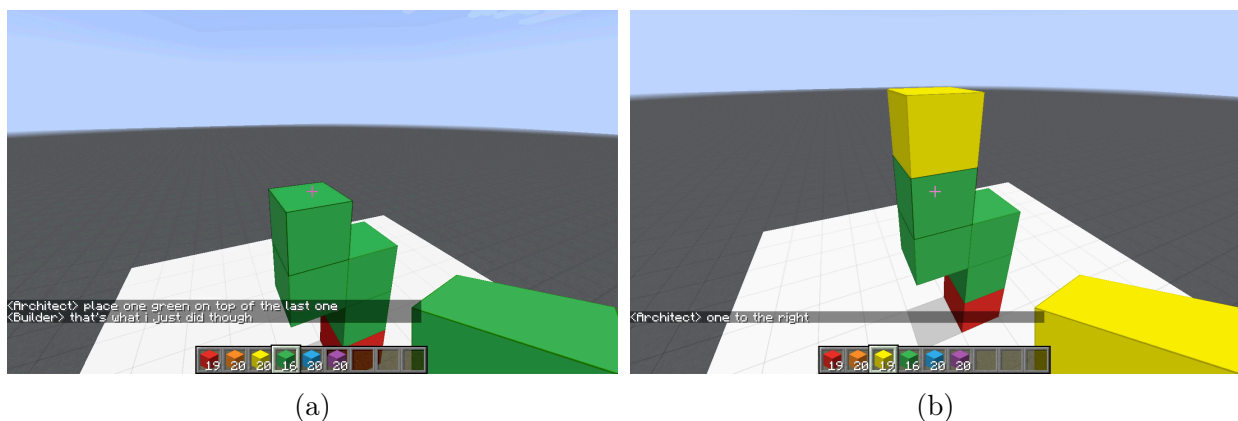


Figure 6.13

A: *place one green on top of the last one*

B: *that's what i just did though*

A: *place one yellow on top of that*

Figure 6.14 summarizes the remainder of the game before **B** decides to quit. **A** continues to issue a meandering list of block-wise instructions that are unintuitive with respect to the target structure, while **B**'s repeated attempts to communicate with **A** are ignored. The model also strictly follows a pattern of generating *up* and *right* relations, without regard for instruction correctness, until **A**'s instructions eventually hit a wall enforced by the dialogue manager's strict error checking. As a result, **A** subjects **B** to an unengaging and frustrating collaborative experience:

A: *one to the right*

B: *what about underneath it?*

A: *one on top of that*

B: *what is this thing?*

A: *place one yellow on top of that*

B: *hello???*

A: *one to the right of the one you just placed*

A: *place a green block on top of the one you just placed*

A: *sorry, please undo that action*

A: *and one to the right of the one you just placed*

...

Discussion While Example 6.1 showcases some of the strengths of our interactive Architect, Example 6.2 highlights some painfully obvious weaknesses. **A** routinely ignores **B**'s attempts to establish contact and blathers on with repetitive-sounding instructions (though this may be mitigated somewhat by instead using a model conditioned on **A** dialogue acts). Furthermore, our model's inability to describe spatial relations other than *up* and *right* preclude it from ever being able to finish the structure, as it becomes impossible to refer back to the incomplete regions at the bottom of the structure. A strict adherence to block-by-block instructions combined with the lack of ability to refer to blocks other than the last **B** action makes for a bland, unintuitive experience in which **B** has no choice other than to slowly and blindly slog through **A**'s instructions. In contrast, we compare this to a snippet from a human-human dialogue from our corpus for the same structure:

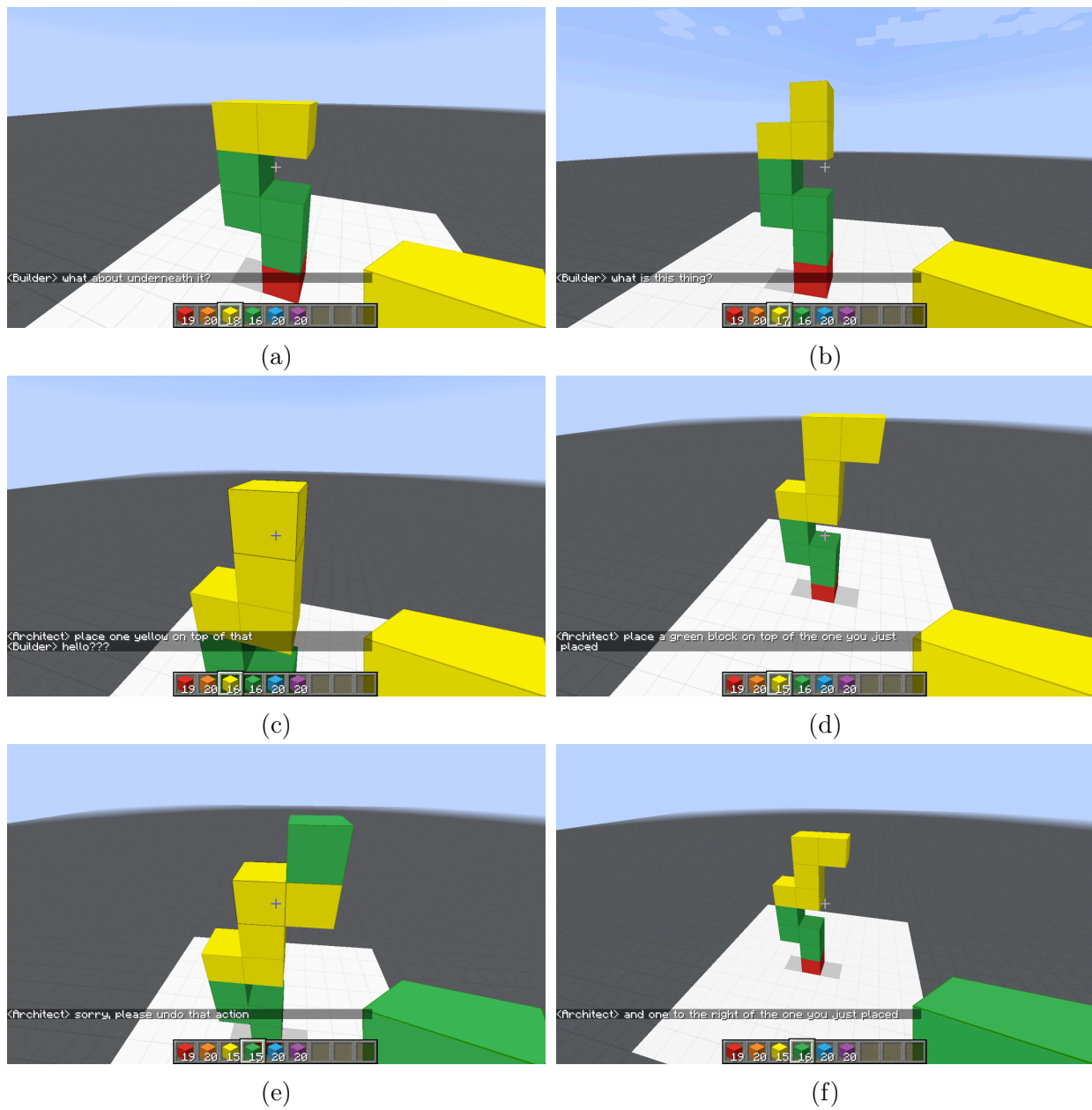


Figure 6.14

A: *this one is all in one vertical plane*

A: *so you'll be building up. Start with two red blocks one space apart*

B *places a red block*

A: *somewhere in the middle*

B *places a red block*

A: *on top of that a row of 7 forest green blocks.*

B *places a green block*

B: *on top of both?*

B *places 3 green blocks*

A: *not vertically, horizontally*

B *removes 3 green blocks and places a green block*

B: *like this?*

A: *and it will be symmetric*

B *places 2 green blocks*

A: *and yeah@!!*

B *places 3 green blocks*

A: *awesome. on top of that, 5 forest green blocks*

B: *are we making a tree?*

B *places 5 green blocks*

A: *kind of. do another layer of the tree (3) then it will change after that*

B *places 3 green blocks*

A: *it kind of looks like a man in a skirt*

A: *on top of that, three light/puke green blocks*

B *places 3 yellow blocks*

...

In a fewer number of dialogue turns, the human **A** was able to instruct **B** to construct the entire bottom half of the structure involving red and green blocks. The interaction here is much more fluid, and **A** is able to not only respond to **B**'s repeated queries appropriately, but also provide intuitive, multi-step instructions that keep **B** engaged in the task. And despite entering into error states due to ambiguous instructions, **A** is able to recover from errors gracefully with appropriate and meaningful corrections that go beyond simple block-wise undo functions. While human-human interaction sets an incredibly high bar for our interactive models, it is this type of fluid interaction that we should strive towards. This speaks to the need for interactive human evaluation of our models and the addition of dialogue-level metrics, such as monitoring user satisfaction, interaction length, and number

of times a Wizard (in this case, our dialogue manager) needs to intervene to correct the model’s mistakes.

Lastly, it is important to note that Examples 1 and 2 feature target structures that are relatively simpler than the typical structures in the Minecraft Dialogue Corpus. Most complex structures consist of blocks that are not immediately adjacent to each other and thus require **A** to reason about and describe gaps, distances, and relations between larger substructures. Also, structures are not always planar, requiring **A** to instruct **B** to move forwards and backwards through space in addition to up, down, left, and right (which our current model already has trouble distinguishing between even without the added complexity). Furthermore, we explicitly avoid structures with floating blocks, a key feature that adds to instruction planning complexity, as both our model and our dialogue manager are not yet equipped to handle them. While a robust interactive Architect should be able to tackle these more complex target structures, a necessary first step is to develop models that can first achieve some degree of competency on easier structures such as those in Examples 1 and 2. To this end, we discuss various avenues of future work that might help our models achieve this in the next chapter.

6.3 CONCLUSION

In this chapter, we described our design of a rule-based dialogue manager used to invoke the block counters v2 Architect utterance model in real interactive scenarios involving human Builders. In addition to prompting the model to generate utterances at appropriate intervals, the dialogue manager is responsible for a number of auxiliary functions, such as identification and graceful recovery from errors and avoiding repetitive outputs, that are crucial to a smooth interactive experience. By walking through examples of our model’s responses in interactive scenarios, we note the strengths and weaknesses of our model in practice and highlight necessary directions for future work. In the final chapter, we dive into these avenues in more detail.

Chapter 7: Conclusion

This thesis explores building interactive agents that can successfully communicate with humans about the physical world around them to collaboratively solve tasks. We have investigated this by defining a novel and challenging instantiation of a situated dialogue task, the Minecraft Collaborative Building Task, in Chapter 1. Furthermore, we collected an associated dataset, the Minecraft Dialogue Corpus, consisting of 509 game logs of humans accomplishing this task, which we described in detail in Chapter 2. In building models for the subtask of Architect utterance generation, our work represents a case study on how far we can push end-to-end models for grounded dialogue tasks with limited data.

In Chapter 3, we defined a baseline sequence-to-sequence model for generating Architect utterances in an offline setting, as well as ways to evaluate this model using an estimated measure of task success (term precision and recall). Here, we showed that conditioning our model on a simple representation of the world, which involves accumulating block counters over global and local regions in the world, gave our model improved ability to generate correct instructions. We further showed that focusing the world representation elsewhere using oracle next Builder action information, akin to adding an explicit instruction planning component to the Architect, resulted in improved performance. In contrast, attempting to learn this world representation in an end-to-end fashion using a richer convolutional neural network, as we investigated in the latter section of Chapter 5, is not fruitful.

We further investigated adding linguistic structure to our corpus and models using Architect dialogue acts in Chapter 4. By defining a coarse set of 6 Architect dialogue acts that represent the most common functions of Architect utterances, we showed that not only can these dialogue acts be modeled with simple multilabel classifiers, but also that adding this additional linguistic structure to our baseline Architect utterance models was beneficial. We further demonstrated that combining ground truth Architect dialogue act information with oracle next Builder actions resulted in the largest gain by far, without requiring any modification to the simple world state representation originally defined in Chapter 3.

In Chapter 5, we explored another way of adding additional structure to Minecraft worlds in our corpus by pretraining convolutional neural models on synthetically generated data to identify the locations of elementary shapes. We then demonstrated preliminary results of applying these pretrained models to real Minecraft target structures.

Finally, in Chapter 6, we returned to the full interactive Collaborative Building Task by designing a rule-based dialogue manager that invokes our block counters Architect utterance model in a real-time game context. By analyzing our model’s performance in interactive sce-

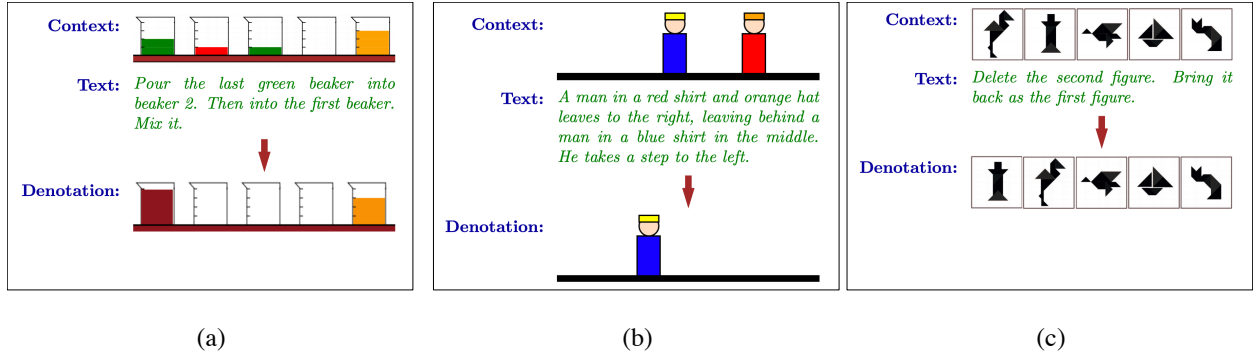


Figure 7.1: Example sequential instructions in SCONE [91] in its three domains: (a) ALCHEMY, (b) SCENE, and (c) TANGRAMS. The domains are designed to feature context-dependent linguistic phenomena, such as ellipsis, action coreference, and object coreference.

narios with human Builders, we highlighted cases where our system was able to successfully complete simple target structures, as well as investigated areas of weakness where our system floundered. In Section 7.2, we use these insights to propose additional areas for future work.

In summary, we have seen that as we tackle more complicated aspects of grounding in dialogue, the more necessary it becomes to have richer representations of these aspects in our models. In particular, situated dialogue requires aligning the world to the current dialogue state, while instruction giving requires aligning the current world to a goal to obtain a meaningful representation of what needs to be done. However, learning these richer representations directly from real, complex, limited human interactions has proven to be a challenge that, in our experience, has been best mitigated not by designing richer end-to-end neural representations that involve more parameters and are more difficult to train, but rather by adding meaningful structured information that we can glean from modular components dedicated to other, smaller subtasks.

7.1 COMPARISON TO CONCURRENT WORK

There have been multiple lines of related work that have emerged concurrently with the development of our work. In this section, we outline some of these related advancements and show how our contributions differ.

Situated language The Sequential CONtext-dependent Execution dataset (SCONE) [91] (Figure 7.1) defines tasks in three domains requiring context-dependent sequential instruction understanding, in which a system is given a world containing several predefined objects and properties and has to predict the final world state by parsing instructions to intermediate

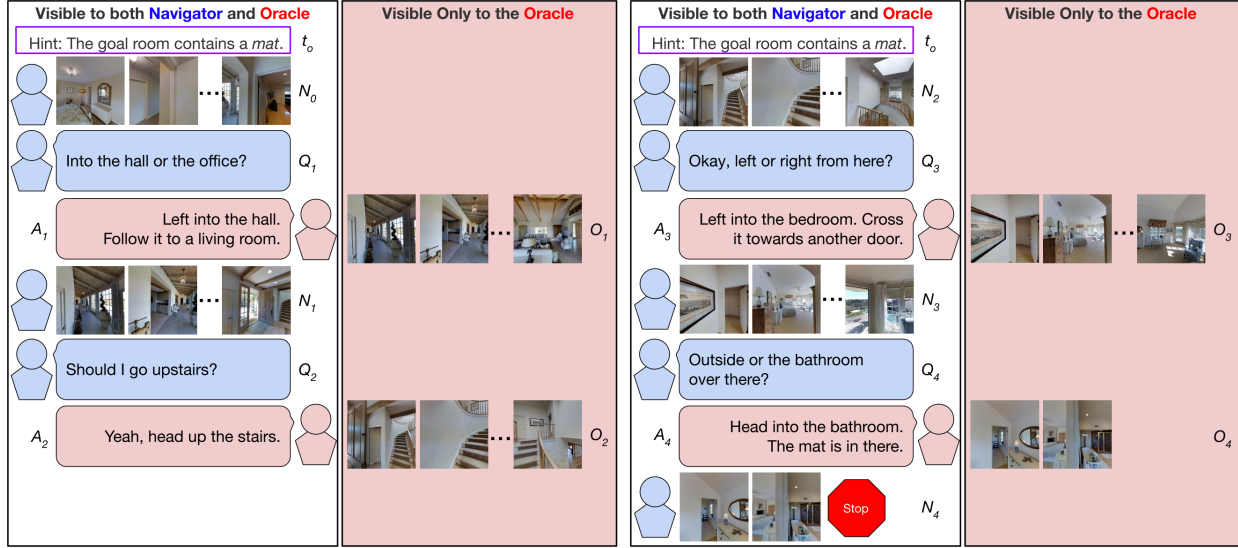


Figure 7.2: Example dialogue from the CVDN corpus [99]. The *Navigator* (*N*) moves through a simulated environment with the guidance of an *Oracle* (*O*), and the two speakers interact via question-answer pairs.

logical forms. An alternative approach to semantic parsing is to map instructions directly to executable actions. This has been explored in various one-way communication scenarios, such as in simplified real-time strategy (RTS) games in which only the follower can execute instructions given by an instructor [92], as well as in collaborative games such as in CEREALBAR [93] where both players can execute actions in a simulated world. Some papers have also directly applied neural action prediction models [94, 95] to SCONE. Situated collaboration has also been explored in two-way dialogue scenarios, such as that of CoDraw [96], where a Drawer communicates with a Teller to reconstruct a described scene, or the Meet Up! corpus [97], where players coordinate and independently navigate to a specific location on a game board. More recently, Vision-and-Language Navigation (VLN) [98], and its dialog counterpart, Cooperative Vision-and-Dialog Navigation (CVDN) [99] (Figure 7.2), focus on instruction following and cooperative interactions in photorealistic navigation settings. In VLN and CVDN, embodied agents situated in a simulated home environment must ground instructions to entities in their field-of-view in order to complete a navigation task.

Since at the time of writing our dataset does not contain any logical forms, we also cannot use semantic parsing approaches, and have to resort to neural action prediction models.¹ However, Minecraft instructions are more challenging than the SCONE tasks because our ac-

¹AMRs for our dialogues are now becoming available [40], and the automatic translation of these annotations to logical forms and resulting executable action sequences is in progress. In the future, the SCONE approach might become viable, albeit our Blocks World scenario is still much more challenging.

“On the (new) fourth tower, mirror Nvidia with UPS.”

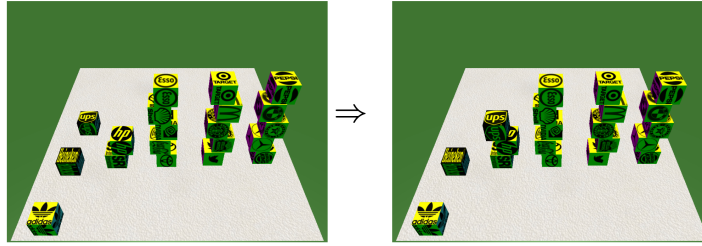


Figure 7.3: Example language instruction in the latest version of the ISI Language Grounding corpus [100].

tion space is significantly larger and our utterances are more complex. Additionally, contrary to the instructions in SCONE, we cannot assume that actions to be executed in Minecraft are described in the last utterance. Minecraft dialogues are also more complex than those in CVDN, because they contain more turns, communication is asynchronous, and our dialogues do not follow the strict question-answer format used in CVDN dialogues. Moreover, construction differs fundamentally from navigation in that construction dynamically changes the environment. While referring expressions in navigation can be safely assumed to refer to objects that exist in the world, construction instructions frequently refer to objects that need to be built by the agent, resulting in more complex utterances that necessarily refer to entities that do not yet exist. And although more recent navigation tasks require real vision, their underlying world state space (as defined by fixed viewpoints and the underlying navigation graph) is just as highly discretized. Instruction execution in our Minecraft scenario does not require vision, but poses an arguably more challenging planning problem with a much larger action space.

Blocks World Apart from our work, there has been a concurrent resurgence of interest in Blocks World-like scenarios. Voxelurn [101] interfaces with human users and learns to understand descriptions of voxel structures of increasing complexity. Human users define 3D voxel structures via a highly programmatic natural language. The interface learns to understand descriptions of increasing complexity, but does not engage in a back-and-forth dialogue with the user. Most closely related to our work are the ISI Language Grounding corpora [100, 102, 103], which feature pairs of scenes involving simulated, uniquely labeled, 3D blocks annotated with single-shot instructions aimed at guiding an (imaginary) partner on how to transform an input scene into the target. In their scenario, the building area is always viewed from a fixed bird’s-eye perspective. Simpler versions of the data retain the grid-based assumption over blocks, and structures consist solely of numeric digits procedu-



(a) A human user instructs a builder bot using the CraftAssist platform [104].

(b) An architect agent instructs a human user using the MC-Saar-Instruct platform [105].

Figure 7.4

rally reconstructed along the horizontal plane. Later versions increase the task complexity significantly by incorporating human-generated, truly 3D structures and removed the grid assumption, as well as allowing for rotations of individual blocks (Figure 7.3). Their blocks behave like physical blocks, disallowing structures with floating blocks that are prevalent in our data. Our work differs considerably in a few other aspects: our corpus features two-way dialogue between an instructor and a real human partner; it also includes a wide range of perspectives as a result of using Minecraft avatars, rather than a fixed bird’s-eye perspective; and we utilize blocks of different colors, allowing for entire substructures to be identified (e.g., “the red pillar”). Furthermore, because our scenario is designed to be asynchronous and allows for mistakes to be made and rectified during the building process, we do not have a clean one-to-one mapping of instructions to action sequences, and the resulting dialogues are much more varied and complex.

Minecraft Combining semantic parsing with simulated human-robot interaction, Facebook CraftAssist [104, 106, 107] is a recent dialogue-enabled framework with an associated dataset for semantic parsing of instructions in Minecraft (Figure 7.4a). Their setup enables two-way human-bot interactions in which a human architect can direct an automated builder using natural language to build complex structures. To bootstrap a semantic parser for instructions, they synthetically generate (using a hand-defined grammar) and crowdsource natural language instructions paired with logical tree structures consisting of action primitives. However, our work addresses a more challenging task in that our data is sourced from human-human dialogues; instructions are more ambiguous, dialogues have larger variety and Builder action sequences are noisier.

MC-Saar-Instruct [105] is another recent platform for instruction-giving agents built on top of Minecraft (Figure 7.4b). MC-Saar-Instruct is an in-house distributed platform on top of the native Minecraft client that connects human users to architect agents in a shared server. In this setup, an architect can be an automated instruction-giving agent or another human player posing in a Wizard-of-Oz setup. In related work, Köhn and Koller (2019) [108] formalize the problem of generating indefinite referring expressions to Minecraft objects that do not yet exist and formulate an algorithm to generate such expressions in the context of building structures in Minecraft games. Köhn et al. (2020) [109] then explore generating instructions in Minecraft at varying levels of abstraction, making use of construction plans for Minecraft structures [110], via hierarchical planning. Their system combines a hierarchical planning system based on Hierarchical Task Networks (HTN) [111, 112] with a chart-based generation system to generate instructions. Our work differs from theirs mainly in that we investigate instruction-giving in Minecraft in a two-way dialogue scenario, and that we explore end-to-end neural approaches, rather than symbolic approaches, for learning to build a variety of unseen Minecraft structures.

7.2 FUTURE WORK

Our work in this thesis just begins to scratch the surface of possibilities with the Collaborative Building Task. Here, we outline various avenues for possible future work.

Collecting more data One of the drawbacks of the Minecraft Dialogue Corpus is simply its limited size. Even with 3 unique dialogues for each target structure, these dialogues can vary wildly depending on the users who assume the roles of **A** and **B**. The dialogues and utterances grow even more complicated for more complex target structures. As neural approaches are known to be data-hungry, one obvious extension would be to collect more dialogues for both existing target structures as well as newly-designed ones. One limitation of the current implementation of our data collection platform that inhibits this is that it must be run in-house with modified Minecraft clients; one workaround would be to investigate better ways of deploying these clients, such as through Docker or a web server (similar to the server-client architectures designed in CraftAssist [104] or MC-Saar-Instruct [105]). Such an architecture would also improve the pipeline for crowdsourcing human evaluations.

As collecting full dialogues can be expensive, another possible option is to crowdsource additional data at the utterance level using the Fixed Viewer screenshots that accompany our existing game logs. While data collected in this way would not be true two-way dialogue data, it may still be helpful to augment existing game logs with such utterance paraphrases.

Data augmentation Another area for future work involves investigating how to properly leverage automated data augmentation techniques to train more robust Architect models. In Jayannavar et al. (2020) [38], we are able to successfully leverage augmented data to train richer models for the Builder Action Prediction (BAP) task. However, we have not been able to achieve any increased gains when training our Architect models in a similar manner. It is unclear why this occurs; one possibility is that our current data augmentation scheme lacks sufficient variety for Architect utterances, which is less of a concern for the BAP task. Regardless, richer representations with larger numbers of trainable parameters can only benefit from training on larger amounts of data, and it would be useful to figure out how to properly take advantage of this for Architect utterance models.

Leveraging AMR annotations for Minecraft dialogues Recently, annotations for our dialogues using Abstract Meaning Representation (AMR) have been released [40]. These annotations contribute a large amount of structured annotation to our dialogues that go far beyond the simple dialogue act labels we defined in Chapter 4. In particular, these annotations provide a detailed and explicit representation of spatial relations and spatial frameworks for entities, such as the Builder, blocks on the grid, and the grid itself. Future work should consider how these richer, structured representations (and in particular, the annotations for the Builder’s frame of reference) could be used for both understanding and generating situated language in Minecraft dialogues.

Training regimes and loss functions Throughout this thesis, we have only considered training our Architect utterance models using cross-entropy loss between predicted and ground truth tokens with teacher forcing, as is standard for NLG tasks. However, training in this manner does not impart any specific knowledge of the overall task goals. In the past, we have tried using alternative loss functions, such as margin ranking loss, in combination with cross-entropy loss to bias the model towards generating utterances that contain tokens instrumental to task success (e.g., the correct colors). However, these experiments proved unfruitful. Future work should investigate how to change the current training scheme to involve more than just cross-entropy loss to better enforce aspects of task success during the model training process.

Integrating with a Builder The current Architect utterance models are responsible for both planning the next sequence of **B** steps as well as describing them in natural language in an end-to-end fashion, and must manage both of these tasks from a single representation that compares *Built* and *Target*. In Chapters 3 and 4, we alluded to the use of an external

instruction planning component by integrating oracle next **B** actions into our block counters models. Future work can expand this avenue further by modularizing the Architect into a system that consists of independent instruction planning and NLG components. One way this can be achieved is by extending the BAP model to predict sequences of next **B** actions based on both *Built* and *Target*.

Another more distant possibility is to train an Architect and a Builder model jointly using reinforcement learning, in which a Builder model executes generated Architect instructions and provides additional loss in the form of a reward function. However, this requires us to start from baseline models that already have reasonable individual performance. More work is required to develop Architect and Builder models separately before this avenue can be explored.

Leveraging large pretrained language models Recent trends in NLP have opted to use transformers [49], powerful and easily parallelizable neural architectures that make use of self-attention, as an alternative to standard RNN architectures. Large pretrained transformer models, trained on vast amounts of data using semi-supervised techniques and finetuned on specific tasks, have broken records and defined new state-of-the-art results on a variety of task benchmarks. One of the main benefits afforded by these models is the ability to finetune a large pretrained model (such as BERT [113], BART [114], GPT-2 [115] or the most recent GPT-3 [116]) on a small amount of in-domain data to achieve competitive results. Such models have also been successfully extended to handle multimodal inputs, such as the joint image and natural language representations in ViLBERT (Vision-and-Language BERT) [117]. One possible avenue for future work is to explore how to finetune large pretrained transformer models for our own task, which would require additional architecture to incorporate world state information, or to replace our RNN backbone with transformers in order to utilize the self-attention mechanism.

Evaluation metrics As we noted in Chapter 3, BLEU [41], originally used for machine translation, is a less-than-ideal standard metric to use to compare models for task-oriented dialogue models. At the same time, our term-specific precision and recall metrics are very noisy measures of task success and depend heavily on the human **A**’s original utterance in each game context. Furthermore, our analysis of model performance in interactive scenarios in Chapter 6 points to a need for dialogue-level evaluation of our models. A necessary area for future exploration thus includes thoughtfully designing and evaluating additional metrics that ideally rely less on the human **A** utterance as a reference, and captures a level of user satisfaction or task success over complete dialogues.

Appendix A: Example Dialogues from the Corpus

Below, we include two sample dialogues from the corpus along with corresponding screenshots of the target structure to be built. Builder actions are interleaved in the dialogue in bolded italics.

Example A.1. *Chicken or a gun turret.*

A: hi

B: Hello

A: ready to go???

B: Yes I am!

A: this is either a chicken or a gun turret

A: start with blue

B: Excellent

A: put a line of three down

B puts down a row of three blue blocks

A: now parallel to that with one blank space add another line

B: Same color?

A: yep

B puts down a row of three blue blocks

A: now put a single block on each middle one

B puts down two blue blocks

A: good. that's the feet/legs

A: now orange

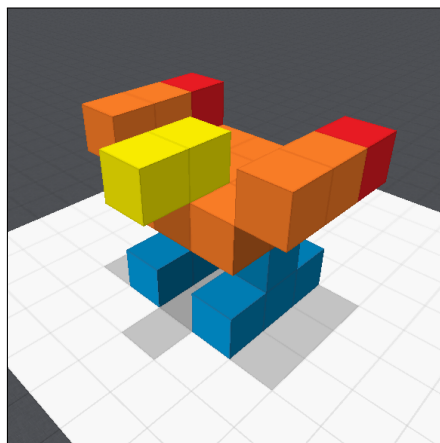


Figure A.1: Target structure for Example A.1.

B: Okay

A: a flat 3x3 square centered over the blue parts

B puts down a square of nine orange blocks

B: Like this?

A: great

A: now wings. two orange blocks in a row up and diagonal away from the body

B puts down two orange blocks, then removes the first

B: Is this the right place?

A: yep. now one more closer to you

B puts down three and removes two orange blocks

A: great now mirror it

B: Copy that

B builds a mirror of the existing wing

A: now a single red block behind each of the wings. still in line with the oranges

B puts down two red blocks

A: then a row of two yellow blocks on the orange side of the wings for the neck/head

A: juts out over the edge

A: parallel to the wings

B: Should they be right next to the wings?

A: centered. one block in the middle edge orange

A: then another block coming away from the body

B puts down two yellow blocks

B: Here?

A: awesome

B: Cool

A: that's it

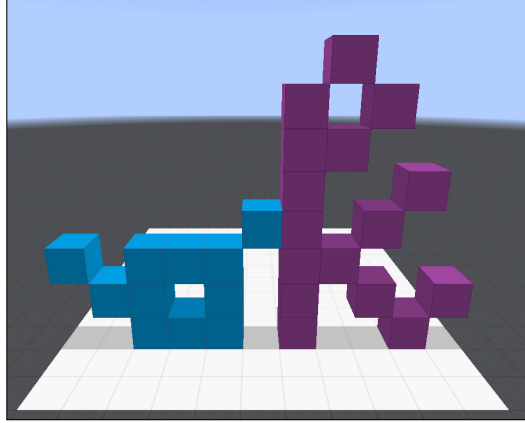


Figure A.2: Target structure for Example A.2.

Example A.2. *Cursive ok.*

B: what is this

A: It looks weird, so I don't really know.

B: ok

B: then guide me blok by block

A: Ok, starting on an edge, make a purple block two blocks in

B puts down a purple block, picks it up, and places it again

B: I'll stay with this perspective

B: unless you want me to change

A: Sorry, one block closer to the edge

B removes and places the purple block

A: Can you make a block 1 off the ground on the edge

B: like on top of that purple block?

A: Like a v with the purple block as the poiunt

B creates a v-shape using floating blocks

B: yeha?

A: Like that, except perpendicular to the edge

B begins to correct the orientation of the v, using a series of block placements and removals

B: there?

A: Yup, can you extend the block on the right diagonally up?

B continues the v to the right

B: like that

A: Yes, can you make a 7 block tall pillar to the right of that block. It should start from the ground

B: what color?

B: purple?

A: Purple still

B builds a pillar of 7 purple blocks

A: Can you extend the top block 2 blocks wtowards the edge

B: which edge

A: The side with the v

B puts down two purple blocks

A: Can you add a block on top and below the middle block the blocks we just placed?

B puts down two purple blocks

B: um

A: Yes?

B: is this it?

A: Oh, yeah, can you punch out the middle block

B removes a purple block

A: Cool, from the block 3 blocks below that, can you extend it two blocks diagonally towards the closest edge?

B: what

B: literally what

A: Sorry,

A: You know how we have the block next to the pillar?

A: Close to the bottom

B: yes

B: the one i am looking at?

A: Can you add a block to it going diagonally.

B: down?

B builds a floating purple block

B: like that

A: One beloow the block you just placed, sorry.

B puts down a purple block, then removes the previous purple block

A: And another block to the left and up

B builds a floating purple block

A: Ok, that is all of the purple blocks. We are going to use blue blocks now.

A: to the right of the pillar,

B: is there symmetry?

A: No,

B: sad

A: I know, maybe it will make sense to you. There is a block attached the the pillar on the fourth block from the ground

B puts down a blue block

B: there

A: Perfect

A: From the right of that, can you make a 3 block pillar from the ground

B builds a pillar of three blue blocks

B: is this a 2D structure?

A: Yes

A: Can you make a ring using the pillar we just made?

B: 3x3

B: ?

A: Yes

A: sorry,

B puts down five blue blocks

B: that

A: Yup, on the middle block of the ring's right side, can you put a blue block?

B puts down a blue block

A: And up and to the right of that, one more blue block

B builds a floating blue block

B: it looks like a cursive "ok" form the other end

A: That should be it. Oh, I never learned cursive.

Appendix B: Term-Specific Word Lists

We use the following word lists to compute term-specific metrics:

- **Colors:** red, orange, yellow, purple, green, blue
- **Spatial relations:** top, right, left, side, middle, up, down, bottom, towards, center, above, diagonal, out, front, here, away, diagonally, behind, back, between, below, vertical, long, tall, sides, flat, touching, high, facing, under, directly, opposite, toward, parallel, standing, near, forward, wide, horizontal, face, underneath, closest, across, perpendicular, rightmost, closer, along, leftmost, vertically, looking, around, whole, centered, degrees, extending, 90, 2d, before, sticking, topmost, edges, adjacent, mirror, perspective, attached, upside, highest, height, touch, upwards, hanging, straight, higher, big, shifted, inside, lower, horizontally, connecting, reference, orientation, upper, upright, inner, stacked, length, longer, apart, small, symmetric, furthest, float, upward, ahead, farthest, hole, hang, outward, angle, faces, short, 180, shorter, oriented, entire, outer, outside, outwards, overhanging, taller, symmetrical, jutting, beneath, inward, inwards, 3d, diagonals
- **Dialogue:** ?, ok, place, put, okay, make, good, sorry, yes, build, another, other, same, add, perfect, yeah, great, no, next, first, remove, last, done, yep, not, cool, nice, placed, stack, move, delete, yup, hello, hi, again, alright, connect, starting, ready, making, break, please, bad, extend, fill, yea, use, check, rid, ya, sure, awesome, correct, gotcha, repeat, leave, exactly, connected, yay, switch, keep, nah, shift, hey, enough, fine, thanks, complete, stand, replace, almost, excellent, oops, rotate, wrong, nope, leaving, punch, continue, finish, sweet, whoops, additional, mistake, placing, removed, final, thank, copy, turn, create, once

Appendix C: Synonym Map

The synonym map used for generating additional references using synonym replacement is as follows:

Word	Synonyms
two	2
three	3
four	4
five	5
six	6
seven	7
eight	8
nine	9
second	2nd
2nd	second
third	3rd
3rd	third
okay	ok
ok	okay
put	place, add
place	put, add
add	put, place
placed	put, added
make	build
build	make
made	built
built	made
building	making
yes	yep, ya, yeah, yup, yea
yep	yes, ya, yeah, yup, yea
ya	yep, yes, yeah, yup, yea
yeah	yes, yep, ya, yup, yea
yup	yes, yep, ya, yeah, yea
yea	yes, yep, ya, yeah, yup
ground	floor

Table C.1: Synonym map

Word	Synonyms
perfect	good, great, awesome, nice, cool, alright
great	good, perfect, awesome, nice, cool, alright
nice	good, great, awesome, perfect, cool, alright
cool	good, great, awesome, perfect, nice, alright
awesome	good, perfect, great, nice, cool, alright
alright	good, perfect, great, nice, cool, awesome
but	however
however	but
start	begin
begin	start
starting	beginning
beginning	starting
remove	delete, break
delete	remove, break
break	remove, delete
floating	hovering
hovering	floating
towards	toward
toward	towards
looks	seems
brick	block
block	brick
bricks	blocks
blocks	bricks
ones	blocks, bricks
done	finished
move	shift
shift	move
spaces	squares
squares	blocks, bricks
think	believe
believe	think
gap	space
shape	structure
hello	hi
hi	hello

Table C.1: Synonym map

Word	Synonyms
empty	blank
blank	empty
below	under, underneath
under	below, underneath
underneath	below, under
tower	stack, pillar
pillar	stack, tower
connect	join
join	connect
connected	joined
joined	connected
level	layer
u	you
you	u
tall	high
high	tall
lol	haha
haha	lol
wait	stop
stop	wait
staircase	stairway
stairway	staircase
want	need
need	want
job	work, stuff
please	pls
pls	please
perpendicular	orthogonal
orthogonal	perpendicular
correct	right
gotcha	understood
understood	gotcha
repeat	redo, mimic
redo	repeat, mimic
mimic	redo, repeat
sort	kind

Table C.1: Synonym map

Word	Synonyms
kind	sort
table	desk
desk	table
problem	worries, prob
worries	problem, prob
prob	problem, worries

Table C.1: Synonym map

Appendix D: Hyperparameter Configurations

For all models, we use 300-dimensional Gated Recurrent Units (GRUs) [48] for all RNN modules and use 300-dimensional pretrained GloVe word embeddings [46].

D.1 BLOCK COUNTERS

This section details the specific hyperparameter configurations used in Section 3.5.

Ablation study hyperparameters For the ablation study, we analyze the effect of adding various block counters representations to a fixed dialogue history representation. Thus, the RNN framework we use for all models in the ablation study is a 2-layer bidirectional encoder RNN connected to a single-layer decoder RNN. All models were trained with dropout of 0.5 for both the RNNs and the counter embedding layers. Specific hyperparameter configurations for the models in Table 3.1 are as follows:

- **seq2seq** (row 1): The baseline sequence-to-sequence model conditioned only on full dialogue history features the abovementioned RNN framework.
- **+ global** (row 2): The model incorporating only the global block counters features a counter embedding size of 15.
- **+ local** (row 3): The model incorporating only local block counters features a counter embedding size of 200.
- **+ both** (row 4): Our final model concatenates both local and global counters and embeds them into a 200-dimensional vector.

Test set hyperparameters We optimize the seq2seq and full models by performing a grid search over model hyperparameters. Specific hyperparameter configurations for the models in Table 3.2 are as follows:

- **seq2seq** (row 1): The baseline model features a single-layer bidirectional encoder RNN connected to a single-layer decoder RNN with a dropout of 0.5 for both RNN modules.
- **full model** (row 2): Our final model features a counter embedding size of 200. It was trained using dropout=0.5 for the counter embedding layers and dropout=0 for RNN modules.

D.2 BLOCK COUNTERS V2

This section details the specific hyperparameter configurations used in Section 3.7.

Ablation study hyperparameters For the ablation study, we analyze the effect of bug fixes and data cleaning by re-running a grid search over block counters model hyperparameters. All models featured single-layer decoder RNNs and were trained with dropout of 0.5 for both the RNNs and the block counter embedding layers. Specific hyperparameter configurations for the models in Table 3.8 are as follows:

- **Baseline** (row 1): The baseline block counters model has the same hyperparameter configuration and model architecture as described in the final ablation study model of Appendix D.1: the global and local counters are concatenated and embedded into a 200-dimensional vector via a single feedforward layer, and the dialogue history is encoded using a 2-layer bidirectional encoder RNN.
- **+ bug fixes** (row 2): The global counters are embedded separately into an 18-dimensional vector via a single feedforward layer, and the local counters are embedded into a 200-dimensional vector using another feedforward layer. The resulting embeddings are concatenated and fed into the decoder RNN, alongside the word embeddings, at every time step. The dialogue history is encoded using a 2-layer bidirectional encoder RNN.
- **+ cleaned data** (row 3): This model uses the same block counters architecture as the model in row 2. However, the dialogue history is encoded using a 1-layer bidirectional encoder RNN.
- **+ constraints** (row 4): The same model from row 3, but with constrained decoding applied during generation.

Test set hyperparameters We optimize the baseline and updated block counters models by performing a grid search over model hyperparameters. Specific hyperparameter configurations for the models in Table 3.9 are as follows:

- **Baseline** (row 1): The baseline block counters model has the same hyperparameter configuration and model architecture as described in the final test set model of Appendix D.1, using a counter embedding size of 200 (dropout=0.5 for counter embedding layers) and a 2-layer bidirectional encoder RNN (dropout=0 for RNN modules).

- **Block counters v2** (row 2): Our final model has the same hyperparameter configuration as that of the ablation study (rows 3 and 4). We found that finetuning dropout values did not produce better results, resulting in a final dropout of 0.5 for both counter embedding and RNN modules.

D.3 USING ORACLE NEXT ACTIONS

The specific hyperparameter configurations used in Section 3.8 are as follows:

- **Block counters v2:** This model uses the same hyperparameter setup as described in Appendix D.2, in which the dialogue history is encoded using a 1-layer bidirectional encoder RNN, and the global and local counters are embedded separately into 18- and 200-dimensional vectors, respectively. The counters embeddings are concatenated and fed into the decoder RNN alongside the word embeddings at every time step.
- **Next actions:** The dialogue history is encoded using a 1-layer bidirectional encoder RNN. The global counters are embedded separately into a 15-dimensional vector via a single feedforward layer, while the local counters (with updated cell representation) are embedded into a 300-dimensional vector using another feedforward layer. The resulting embeddings are concatenated to the decoder’s 300-dimensional output at every time step to produce a 615-dimensional representation which is then embedded using a 2-layer feedforward network. The final 615-dimensional embedding is then mapped to the output vocabulary size using a final linear layer.

D.4 DIALOGUE ACTS FOR ARCHITECT UTTERANCE GENERATION

This section details the specific hyperparameter configurations used in Section 4.5. Specific hyperparameter configurations for the models in Tables 4.6 and 4.7 are as follows:

- **Block counters v2** (row 1): The block counters model we use as a baseline has the same hyperparameter configuration and model architecture as described in the final model of Appendix D.2: the dialogue history is encoded using a 1-layer bidirectional encoder RNN, and the global counters are embedded separately into an 18-dimensional vector via a single feedforward layer, and the local counters are embedded into a 200-dimensional vector using another feedforward layer. The resulting embeddings are concatenated and fed into the decoder RNN, alongside the word embeddings, at every time step.

- **Ground truth train/test** (row 3): the dialogue history is encoded using a 2-layer bidirectional encoder RNN, while the global counters are embedded separately into a 15-dimensional vector via a single feedforward layer, and the local counters are embedded into a 100-dimensional vector using another feedforward layer. The dialogue acts are not embedded and are fed into the decoder at every time step in the raw multi-hot representation.¹
- **Ground truth train/predicted test** (row 4): The exact same model as row 2, but uses the thresholded predicted dialogue acts generated by our dialogue act predictor model at test time.
- **Predicted train/test** (row 5): the dialogue history is encoded using a 2-layer bidirectional encoder RNN, while the global counters are embedded separately into a 10-dimensional vector via a single feedforward layer, and the local counters are embedded into a 200-dimensional vector using another feedforward layer. The predicted dialogue acts are thresholded with a value of 0.5, embedded into a 5-dimensional vector representation via a feedforward layer, then fed into the decoder at every time step.²

D.5 COMBINING DIALOGUE ACTS WITH ORACLE NEXT ACTIONS

Specific hyperparameter configurations used in Section 4.6 are as follows:

- **Block counters v2**: This model uses the same hyperparameter setup as described in Appendix D.2, in which the dialogue history is encoded using a 1-layer bidirectional encoder RNN, and the global and local counters are embedded separately into 18- and 200-dimensional vectors, respectively. The counters embeddings are concatenated and fed into the decoder RNN alongside the word embeddings at every time step.
- **Next actions (NA) only**: This model uses the same hyperparameter setup as described in Appendix D.3, in which the dialogue history is encoded using a 1-layer bidirectional encoder RNN, the global counters are embedded separately into a 15-dimensional vector via a single feedforward layer, while the local counters are embedded into a 300-dimensional vector using another feedforward layer. The resulting

¹Through a grid search, we found that embedding the multi-hot dialogue act representation using an additional embedding layer helped models that were trained on predicted acts (row 5), but not ones trained on ground truth acts (row 3).

²We also tried including the vector of predicted probabilities produced by the dialogue act predictor directly as input to the utterance generation model (without thresholding them first), but this did not lead to better performance.

counters embeddings are concatenated to the decoder’s 300-dimensional output at every time step to produce a 615-dimensional representation which is then embedded using a 2-layer feedforward network and mapped to the output vocabulary size.

- **Ground truth dialogue acts (DA) only:** This model uses the same hyperparameter setup as described in Appendix D.4, in which the dialogue history is encoded using a 2-layer bidirectional encoder RNN, while the global counters are embedded separately into a 15-dimensional vector via a single feedforward layer, and the local counters are embedded into a 100-dimensional vector using another feedforward layer. A multi-hot representation of the dialogue act is concatenated to the counter representation and fed to the decoder’s input at every time step.
- **Both next actions and dialogue acts:** The dialogue history is encoded using a 1-layer bidirectional encoder RNN. The global and local counters are embedded separately into 18- and 350-dimensional vectors, respectively. The block counters embeddings are concatenated with a multi-hot representation of the dialogue act and fed into the decoder RNN alongside the word embeddings at every time step.

References

- [1] T. Winograd, “Procedures as a representation for data in a computer program for understanding natural language,” MIT. Cent. Space Res., Tech. Rep., 1971. [Online]. Available: <https://apps.dtic.mil/dtic/tr/fulltext/u2/721399.pdf>
- [2] S. Kim, L. F. D’Haro, R. E. Banchs, J. D. Williams, M. Henderson, and K. Yoshino, “The fifth dialog state tracking challenge,” in *2016 IEEE Spoken Language Technology Workshop, SLT 2016, San Diego, CA, USA, December 13-16, 2016*. IEEE, 2016. [Online]. Available: <https://doi.org/10.1109/SLT.2016.7846311> pp. 511–517.
- [3] S. Kim, L. F. D’Haro, R. E. Banchs, J. D. Williams, and M. Henderson, “The fourth dialog state tracking challenge,” in *Dialogues with Social Robots - Enablements, Analyses, and Evaluation, Seventh International Workshop on Spoken Dialogue Systems, IWSDS 2016, Saariselkä, Finland, January 13-16, 2016*, ser. Lecture Notes in Electrical Engineering, K. Jokinen and G. Wilcock, Eds., vol. 427. Springer, 2016. [Online]. Available: https://doi.org/10.1007/978-981-10-2585-3_36 pp. 435–449.
- [4] P. Budzianowski, T.-H. Wen, B.-H. Tseng, I. Casanueva, S. Ultes, O. Ramadan, and M. Gašić, “MultiWOZ - a large-scale multi-domain wizard-of-Oz dataset for task-oriented dialogue modelling,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018. [Online]. Available: <https://www.aclweb.org/anthology/D18-1547> pp. 5016–5026.
- [5] A. Ritter, C. Cherry, and B. Dolan, “Unsupervised modeling of Twitter conversations,” in *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Los Angeles, California: Association for Computational Linguistics, June 2010. [Online]. Available: <https://www.aclweb.org/anthology/N10-1020> pp. 172–180.
- [6] N. Schrading, C. Ovesdotter Alm, R. Ptucha, and C. Homan, “An analysis of domestic abuse discourse on Reddit,” in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015. [Online]. Available: <https://www.aclweb.org/anthology/D15-1309> pp. 2577–2583.
- [7] R. Lowe, N. Pow, I. Serban, and J. Pineau, “The Ubuntu dialogue corpus: A large dataset for research in unstructured multi-turn dialogue systems,” in *Proceedings of the 16th Annual Meeting of the Special Interest Group on Discourse and Dialogue*. Prague, Czech Republic: Association for Computational Linguistics, Sep. 2015. [Online]. Available: <https://www.aclweb.org/anthology/W15-4640> pp. 285–294.

- [8] A. Das, S. Kottur, K. Gupta, A. Singh, D. Yadav, J. M. Moura, D. Parikh, and D. Batra, “Visual Dialog,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. [Online]. Available: http://openaccess.thecvf.com/content_cvpr_2017/papers/Das_Visual_Dialog_CVPR_2017_paper.pdf pp. 326–335.
- [9] R. Pasunuru and M. Bansal, “Game-based video-context dialogue,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018. [Online]. Available: <https://www.aclweb.org/anthology/D18-1012> pp. 125–136.
- [10] D. Roy and E. Reiter, “Connecting language to the world,” *Artificial Intelligence*, vol. 167, no. 1-2, pp. 1–12, 2005. [Online]. Available: <https://doi.org/10.1016/j.artint.2005.06.002>
- [11] S. Tellex, T. Kollar, S. Dickerson, M. Walter, A. Banerjee, S. Teller, and N. Roy, “Understanding natural language commands for robotic navigation and mobile manipulation,” in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011. [Online]. Available: <https://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3623> pp. 1507–1514.
- [12] J. Thomason, S. Zhang, R. J. Mooney, and P. Stone, “Learning to interpret natural language commands through human-robot dialog,” in *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)*, 2015. [Online]. Available: <https://www.aaai.org/ocs/index.php/IJCAI/IJCAI15/paper/view/10957/10931> pp. 1923–1929.
- [13] D. K. Misra, J. Sung, K. Lee, and A. Saxena, “Tell me Dave: Context-sensitive grounding of natural language to manipulation instructions,” *The International Journal of Robotics Research*, vol. 35, no. 1-3, pp. 281–300, 2016. [Online]. Available: <https://doi.org/10.1177/0278364915602060>
- [14] J. Y. Chai, Q. Gao, L. She, S. Yang, S. Saba-Sadiya, and G. Xu, “Language to action: Towards interactive task learning with physical agents,” in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*. International Joint Conferences on Artificial Intelligence Organization, 7 2018. [Online]. Available: <https://doi.org/10.24963/ijcai.2018/1> pp. 2–9.
- [15] A. Koller, K. Striegnitz, D. Byron, J. Cassell, R. Dale, J. Moore, and J. Oberlander, “The first challenge on generating instructions in virtual environments,” in *Empirical Methods in Natural Language Generation*, E. Krahmer and M. Theune, Eds. Berlin, Heidelberg: Springer-Verlag, 2010. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1880370.1880391> pp. 328–352.
- [16] D. Chen and R. Mooney, “Learning to interpret natural language navigation instructions from observations,” in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011. [Online]. Available: <https://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3701> pp. 859–865.

- [17] S. Janarthanam, O. Lemon, and X. Liu, “A web-based evaluation framework for spatial instruction-giving systems,” in *Proceedings of the ACL 2012 System Demonstrations*. Jeju Island, Korea: Association for Computational Linguistics, July 2012. [Online]. Available: <https://www.aclweb.org/anthology/P12-3009> pp. 49–54.
- [18] “What is Minecraft? | Minecraft,” 2020. [Online]. Available: <https://www.minecraft.net/en-us/about-minecraft/>
- [19] M. Johnson, K. Hofmann, T. Hutton, and D. Bignell, “The Malmö platform for artificial intelligence experimentation,” in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16)*, 2016. [Online]. Available: <https://pdfs.semanticscholar.org/a6c7/836a1877abb13c56d0ec29e9cde0d6e60cf8.pdf> pp. 4246–4247.
- [20] C. Danescu-Niculescu-Mizil and L. Lee, “Chameleons in imagined conversations: A new approach to understanding coordination of linguistic style in dialogs,” in *Proceedings of the 2nd Workshop on Cognitive Modeling and Computational Linguistics*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011. [Online]. Available: <https://www.aclweb.org/anthology/W11-0609> pp. 76–87.
- [21] J. Tiedemann, “Parallel data, tools and interfaces in OPUS,” in *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC’12)*. Istanbul, Turkey: European Language Resources Association (ELRA), May 2012. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2012/pdf/463_Paper.pdf pp. 2214–2218.
- [22] E. Dinan, S. Roller, K. Shuster, A. Fan, M. Auli, and J. Weston, “Wizard of Wikipedia: Knowledge-powered conversational agents,” in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.
- [23] K. Gopalakrishnan, B. Hedayatnia, Q. Chen, A. Gottardi, S. Kwatra, A. Venkatesh, R. Gabriel, and D. Hakkani-Tür, “Topical-Chat: Towards Knowledge-Grounded Open-Domain Conversations,” in *Proc. Interspeech 2019*, 2019. [Online]. Available: <http://dx.doi.org/10.21437/Interspeech.2019-3079> pp. 1891–1895.
- [24] K. Zhou, S. Prabhumoye, and A. W. Black, “A dataset for document grounded conversations,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018. [Online]. Available: <https://www.aclweb.org/anthology/D18-1076> pp. 708–713.
- [25] C.-W. Liu, R. Lowe, I. Serban, M. Noseworthy, L. Charlin, and J. Pineau, “How NOT to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation,” in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016. [Online]. Available: <https://www.aclweb.org/anthology/D16-1230> pp. 2122–2132.

- [26] A. Rastogi, X. Zang, S. Sunkara, R. Gupta, and P. Khaitan, “Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset,” in *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence, (AAAI-20), New York, New York, USA, February 7-12, 2020*, vol. 34, 2020, pp. 8689–8696.
- [27] A. Bordes, Y.-L. Boureau, and J. Weston, “Learning end-to-end goal-oriented dialog,” in *5th International Conference on Learning Representations, ICLR 2017, Palais des Congrès Neptune, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. [Online]. Available: <https://arxiv.org/abs/1605.07683>
- [28] A. Saha, M. M. Khapra, and K. Sankaranarayanan, “Towards building large scale multimodal domain-aware conversation systems,” in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, S. A. McIlraith and K. Q. Weinberger, Eds. AAAI Press, 2018. [Online]. Available: <https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17104> pp. 696–704.
- [29] A. H. Anderson, M. Bader, E. G. Bard, E. Boyle, G. Doherty, S. Garrod, S. Isard, J. Kowtko, J. McAllister, J. Miller et al., “The HCRC map task corpus,” *Language and speech*, vol. 34, no. 4, pp. 351–366, 1991. [Online]. Available: <https://journals.sagepub.com/doi/pdf/10.1177/002383099103400404>
- [30] M. MacMahon, B. Stankiewicz, and B. Kuipers, “Walk the talk: Connecting language, knowledge, and action in route instructions,” in *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2*, ser. AAAI’06. AAAI Press, 2006, p. 1475–1482.
- [31] A. Vlachos and S. Clark, “A new corpus and imitation learning framework for context-dependent semantic parsing,” *Transactions of the Association for Computational Linguistics*, vol. 2, pp. 547–560, 2014. [Online]. Available: <https://www.aclweb.org/anthology/Q14-1042>
- [32] D. Chen and R. Mooney, “Learning to interpret natural language navigation instructions from observations,” in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011. [Online]. Available: <https://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3701> pp. 859–865.
- [33] Y. Artzi and L. Zettlemoyer, “Weakly supervised learning of semantic parsers for mapping instructions to actions,” *Transactions of the Association for Computational Linguistics*, vol. 1, pp. 49–62, 2013. [Online]. Available: <https://www.aclweb.org/anthology/Q13-1005>
- [34] T. Kollar, S. Tellex, D. Roy, and N. Roy, “Toward understanding natural language directions,” in *Proceedings of the 5th ACM/IEEE International Conference on Human-Robot Interaction*, ser. HRI ’10. IEEE Press, 2010, p. 259–266.

- [35] C. Matuszek, E. Herbst, L. Zettlemoyer, and D. Fox, “Learning to parse natural language commands to a robot control system,” in *Proceedings of the 13th International Symposium on Experimental Robotics (ISER)*, J. P. Desai, G. Dudek, O. Khatib, and V. Kumar, Eds., 2013. [Online]. Available: https://doi.org/10.1007/978-3-319-00065-7_28
- [36] J. Thomason, J. Sinapov, M. Svetlik, P. Stone, and R. J. Mooney, “Learning multi-modal grounded linguistic semantics by playing “I Spy”,” in *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI-16)*, New York City, 2016. [Online]. Available: <http://www.cs.utexas.edu/users/ai-labpub-view.php?PubID=127564> pp. 3477–3483.
- [37] J. Thomason, A. Padmakumar, J. Sinapov, J. Hart, P. Stone, and R. J. Mooney, “Opportunistic active learning for grounding natural language descriptions,” in *Proceedings of the 1st Annual Conference on Robot Learning (CoRL-17)*, S. Levine, V. Vanhoucke, and K. Goldberg, Eds. Mountain View, California: PMLR, November 2017. [Online]. Available: <http://www.cs.utexas.edu/users/ai-labpub-view.php?PubID=127657> pp. 67–76.
- [38] P. Jayannavar, A. Narayan-Chen, and J. Hockenmaier, “Learning to execute instructions in a Minecraft dialogue,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, July 2020. [Online]. Available: <https://www.aclweb.org/anthology/2020.acl-main.232> pp. 2589–2602.
- [39] A. Narayan-Chen, P. Jayannavar, and J. Hockenmaier, “Collaborative dialogue in Minecraft,” in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, July 2019. [Online]. Available: <https://www.aclweb.org/anthology/P19-1537> pp. 5405–5415.
- [40] J. Bonn, M. Palmer, Z. Cai, and K. Wright-Bettner, “Spatial AMR: Expanded spatial annotation in the context of a grounded Minecraft corpus,” in *Proceedings of The 12th Language Resources and Evaluation Conference*. Marseille, France: European Language Resources Association, May 2020. [Online]. Available: <https://www.aclweb.org/anthology/2020.lrec-1.601> pp. 4883–4892.
- [41] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, July 2002. [Online]. Available: <https://www.aclweb.org/anthology/P02-1040> pp. 311–318.
- [42] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning Representations by Back-Propagating Errors*. Cambridge, MA, USA: MIT Press, 1988, p. 696–699.

- [43] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: <http://www.deeplearningbook.org>
- [44] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, Nov 1997.
- [45] “Introduction to recurrent neural networks (RNN) with dinosaurs,” 2019. [Online]. Available: <https://towardsdatascience.com/introduction-to-recurrent-neural-networks-rnn-with-dinosaurs-790e74e3e6f6>
- [46] J. Pennington, R. Socher, and C. Manning, “GloVe: Global vectors for word representation,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014. [Online]. Available: <https://www.aclweb.org/anthology/D14-1162> pp. 1532–1543.
- [47] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [48] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” in *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.
- [49] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf> pp. 5998–6008.
- [50] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in *Advances in neural information processing systems*, 2014. [Online]. Available: <http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf> pp. 3104–3112.
- [51] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using RNN encoder–decoder for statistical machine translation,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014. [Online]. Available: <https://www.aclweb.org/anthology/D14-1179> pp. 1724–1734.
- [52] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. [Online]. Available: <http://arxiv.org/abs/1409.0473>

- [53] “Sequence to sequence learning.” [Online]. Available: https://d2l.ai/chapter_recurrent-modern/seq2seq.html
- [54] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, “The curious case of neural text degeneration,” in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: <https://openreview.net/forum?id=rygGQyrFvH>
- [55] J. Li, W. Monroe, and D. Jurafsky, “A simple, fast diverse decoding algorithm for neural generation,” *arXiv preprint arXiv:1611.08562*, 2016.
- [56] O. Dušek, J. Novikova, and V. Rieser, “Evaluating the state-of-the-art of end-to-end natural language generation: The E2E NLG challenge,” *Computer Speech & Language*, vol. 59, pp. 123 – 156, 2020. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0885230819300919>
- [57] A. See, P. J. Liu, and C. D. Manning, “Get to the point: Summarization with pointer-generator networks,” in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, July 2017. [Online]. Available: <https://www.aclweb.org/anthology/P17-1099> pp. 1073–1083.
- [58] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, “Show and tell: A neural image caption generator,” *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3156–3164, 2015. [Online]. Available: https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Vinyals_Show_and_Tell_2015_CVPR_paper.pdf
- [59] A. Karpathy and L. Fei-Fei, “Deep visual-semantic alignments for generating image descriptions,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 4, p. 664–676, Apr. 2017. [Online]. Available: <https://doi.org/10.1109/TPAMI.2016.2598339>
- [60] C. Callison-Burch, M. Osborne, and P. Koehn, “Re-evaluating the role of Bleu in machine translation research,” in *11th Conference of the European Chapter of the Association for Computational Linguistics*. Trento, Italy: Association for Computational Linguistics, Apr. 2006. [Online]. Available: <https://www.aclweb.org/anthology/E06-1032>
- [61] M. Hodosh, P. Young, and J. Hockenmaier, “Framing image description as a ranking task: Data, models and evaluation metrics,” *Journal of Artificial Intelligence Research*, vol. 47, pp. 853–899, 2013. [Online]. Available: <https://www.jair.org/index.php/jair/article/view/10833>

- [62] J. Novikova, O. Dušek, A. Cercas Curry, and V. Rieser, “Why we need new evaluation metrics for NLG,” in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017. [Online]. Available: <https://www.aclweb.org/anthology/D17-1238> pp. 2241–2252.
- [63] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: <http://arxiv.org/abs/1412.6980>
- [64] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterton, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010. [Online]. Available: <http://proceedings.mlr.press/v9/glorot10a.html> pp. 249–256.
- [65] A. F. Hayes and K. Krippendorff, “Answering the call for a standard reliability measure for coding data,” *Communication Methods and Measures*, vol. 1, no. 1, pp. 77–89, 2007. [Online]. Available: <https://doi.org/10.1080/19312450709336664>
- [66] L. Wittgenstein, *Philosophical Investigations*. Blackwell, 1953. [Online]. Available: </bib/wittgenstein/Wittgenstein1953/Ludwig.Wittgenstein.-.Philosophical.Investigations.pdf>
- [67] J. L. Austin, *How to do things with words*. Oxford University Press, 1962.
- [68] J. R. Searle and J. R. Searle, *Speech acts: An essay in the philosophy of language*. Cambridge University Press, 1969, vol. 626.
- [69] D. Jurafsky and J. H. Martin, *Speech and Language Processing (2nd Edition)*. USA: Prentice-Hall, Inc., 2009.
- [70] J. R. Searle, “A taxonomy of illocutionary acts,” in *Language, Mind and Knowledge*, K. Gunderson, Ed. University of Minnesota Press, 1975, pp. 344–369.
- [71] A. Stolcke, K. Ries, N. Coccaro, E. Shriberg, R. Bates, D. Jurafsky, P. Taylor, R. Martin, C. Van Ess-Dykema, and M. Meteer, “Dialogue act modeling for automatic tagging and recognition of conversational speech,” *Computational Linguistics*, vol. 26, no. 3, pp. 339–374, 2000. [Online]. Available: <https://www.aclweb.org/anthology/J00-3003>
- [72] H. Bunt, “Context and dialogue control,” *THINK Quarterly*, vol. 3, 1994.
- [73] R. Power, “The organisation of purposeful dialogues,” *Linguistics*, vol. 17, no. 1-2, pp. 107 – 152, 1979. [Online]. Available: <https://www.degruyter.com/view/journals/ling/17/1-2/article-p107.xml>

- [74] J. Carletta, A. Isard, S. Isard, J. C. Kowtko, G. Doherty-Sneddon, and A. H. Anderson, “The reliability of a dialogue structure coding scheme,” *Computational Linguistics*, vol. 23, no. 1, pp. 13–31, 1997. [Online]. Available: <https://www.aclweb.org/anthology/J97-1002>
- [75] J. Allen and M. Core, “Draft of DAMSL: Dialog act markup in several layers,” 1997. [Online]. Available: <http://www.cs.rochester.edu/research/speech/damsl/RevisedManual>
- [76] J. J. Godfrey, E. C. Holliman, and J. McDaniel, “Switchboard: Telephone speech corpus for research and development,” in *Proceedings of the 1992 IEEE International Conference on Acoustics, Speech and Signal Processing - Volume 1*, ser. ICASSP’92. USA: IEEE Computer Society, 1992, p. 517–520.
- [77] S. Jekat, A. Klein, E. Maier, I. Maleck, M. Mast, and J. Quantz, “Dialogue acts in VERBMOBIL,” no. VM-Report 65, 1995. [Online]. Available: https://publikationen.sulb.uni-saarland.de/bitstream/20.500.11880/25208/1/report_65_95.pdf
- [78] A. Stent, “A conversation acts model for generating spoken dialogue contributions,” *Computer Speech and Language*, vol. 16, pp. 313–352, 2002.
- [79] J. Allen, G. Ferguson, and A. Stent, “An architecture for more realistic conversational systems,” in *Proceedings of the 6th International Conference on Intelligent User Interfaces*, ser. IUI ’01. New York, NY, USA: Association for Computing Machinery, 2001. [Online]. Available: <https://doi.org/10.1145/359784.359822> p. 1–8.
- [80] S. Larsson and D. R. Traum, “Information state and dialogue management in the TRINDI dialogue move engine toolkit,” *Natural Language Engineering*, vol. 6, no. 3–4, p. 323–340, Sep. 2000. [Online]. Available: <https://doi.org/10.1017/S1351324900002539>
- [81] D. R. Traum and S. Larsson, *The Information State Approach to Dialogue Management*. Dordrecht: Springer Netherlands, 2003, pp. 325–353. [Online]. Available: https://doi.org/10.1007/978-94-010-0019-2_15
- [82] E. Levin, R. Pieraccini, and W. Eckert, “A stochastic model of human-machine interaction for learning dialog strategies,” *IEEE Transactions on Speech and Audio Processing*, vol. 8, no. 1, pp. 11–23, 2000.
- [83] C. Lambert, A. Cordes, E. Kaplan, P. Jayannavar, and J. Hockenmaier, “Virtual world context encoding for grounded dialogue in Minecraft,” 2019. [Online]. Available: http://www.dreuarchive.cra.org/2019/Lambert/uploads/1/2/6/0/126008504/dreu_final_report.pdf
- [84] H. Noh, S. Hong, and B. Han, “Learning deconvolution network for semantic segmentation,” in *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ser. ICCV ’15. USA: IEEE Computer Society, 2015. [Online]. Available: <https://doi.org/10.1109/ICCV.2015.178> p. 1520–1528.

- [85] J. Long, E. Shelhamer, and T. Darrell, “Fully convolutional networks for semantic segmentation,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.
- [86] Y. Lecun, *Generalization and network design strategies*. Elsevier, 1989.
- [87] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ser. ICML’10. Madison, WI, USA: Omnipress, 2010. [Online]. Available: <https://icml.cc/Conferences/2010/papers/432.pdf> p. 807–814.
- [88] Y. Zhou and R. Chellappa, “Computation of optical flow using a neural network,” *IEEE 1988 International Conference on Neural Networks*, pp. 71–78 vol.2, 1988. [Online]. Available: <https://ieeexplore.ieee.org/document/23914>
- [89] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: <http://arxiv.org/abs/1409.1556>
- [90] A. Narayan-Chen, C. Graber, M. Das, M. R. Islam, S. Dan, S. Natarajan, J. R. Doppa, J. Hockenmaier, M. Palmer, and D. Roth, “Towards problem solving agents that communicate and learn,” in *Proceedings of the First Workshop on Language Grounding for Robotics*. Vancouver, Canada: Association for Computational Linguistics, Aug. 2017. [Online]. Available: <https://www.aclweb.org/anthology/W17-2812> pp. 95–103.
- [91] R. Long, P. Pasupat, and P. Liang, “Simpler context-dependent logical forms via model projections,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016. [Online]. Available: <https://www.aclweb.org/anthology/P16-1138> pp. 1456–1465.
- [92] H. Hu, D. Yarats, Q. Gong, Y. Tian, and M. Lewis, “Hierarchical decision making by generating and following natural language instructions,” in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019. [Online]. Available: <https://arxiv.org/abs/1906.00744> pp. 10 025–10 034.
- [93] A. Suhr, C. Yan, J. Schluger, S. Yu, H. Khader, M. Mouallem, I. Zhang, and Y. Artzi, “Executing instructions in situated collaborative interactions,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019. [Online]. Available: <https://www.aclweb.org/anthology/D19-1218> pp. 2119–2130.

- [94] A. Suhr and Y. Artzi, “Situating mapping of sequential instructions to actions with single-step reward observation,” in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, July 2018. [Online]. Available: <https://www.aclweb.org/anthology/P18-1193> pp. 2072–2082.
- [95] H. Huang, E. Choi, and W. Yih, “FlowQA: Grasping flow in history for conversational machine comprehension,” in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: <https://openreview.net/forum?id=ByftGnR9KX>
- [96] J.-H. Kim, N. Kitaev, X. Chen, M. Rohrbach, B.-T. Zhang, Y. Tian, D. Batra, and D. Parikh, “CoDraw: Collaborative drawing as a testbed for grounded goal-driven communication,” in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, July 2019. [Online]. Available: <https://www.aclweb.org/anthology/P19-1651> pp. 6495–6513.
- [97] N. Ilinykh, S. Zarrieß, and D. Schlangen, “Meet Up! A corpus of joint activity dialogues in a visual environment,” in *Proceedings of the 23rd Workshop on the Semantics and Pragmatics of Dialogue - Full Papers*. London, United Kingdom: SEMDIAL, Sep. 2019. [Online]. Available: http://semdial.org/anthology/Z19-Ilinykh_semdial_0006.pdf
- [98] P. Anderson, Q. Wu, D. Teney, J. Bruce, M. Johnson, N. Sünderhauf, I. D. Reid, S. Gould, and A. van den Hengel, “Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments,” in *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 2018. [Online]. Available: http://openaccess.thecvf.com/content_cvpr_2018/html/Anderson_Vision-and-Language_Navigation_Interpreting_CVPR_2018_paper.html pp. 3674–3683.
- [99] J. Thomason, M. Murray, M. Cakmak, and L. Zettlemoyer, “Vision-and-dialog navigation,” in *Proceedings of the 3rd Annual Conference on Robot Learning (CoRL-19)*, 2019. [Online]. Available: <https://arxiv.org/abs/1907.04957>
- [100] Y. Bisk, K. Shih, Y. Choi, and D. Marcu, “Learning interpretable spatial operations in a rich 3D Blocks World,” in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018. [Online]. Available: <https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17410> pp. 5028–5036.
- [101] S. I. Wang, S. Ginn, P. Liang, and C. D. Manning, “Naturalizing a programming language via interactive learning,” in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, July 2017. [Online]. Available: <https://www.aclweb.org/anthology/P17-1086> pp. 929–938.

- [102] Y. Bisk, D. Marcu, and W. Wong, “Towards a dataset for human computer communication via grounded language acquisition,” in *AAAI Workshop: Symbiotic Cognitive Systems*, 2016. [Online]. Available: <https://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12652>
- [103] Y. Bisk, D. Yuret, and D. Marcu, “Natural language communication with robots,” in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego, California: Association for Computational Linguistics, June 2016. [Online]. Available: <https://www.aclweb.org/anthology/N16-1089> pp. 751–761.
- [104] J. Gray, K. Srinet, Y. Jernite, H. Yu, Z. Chen, D. Guo, S. Goyal, C. L. Zitnick, and A. Szlam, “CraftAssist: A framework for dialogue-enabled interactive agents,” *arXiv preprint arXiv:1907.08584*, 2019. [Online]. Available: <http://arxiv.org/abs/1907.08584>
- [105] A. Köhn, J. Wichlacz, C. Schäfer, Á. Torralba, J. Hoffmann, and A. Koller, “MC-saar-instruct: a platform for Minecraft instruction giving agents,” in *Proceedings of the 21th Annual Meeting of the Special Interest Group on Discourse and Dialogue*. Online: Association for Computational Linguistics, July 2020. [Online]. Available: <https://www.aclweb.org/anthology/2020.sigdial-1.7> pp. 53–56.
- [106] K. Srinet, Y. Jernite, J. Gray, and A. Szlam, “CraftAssist instruction parsing: Semantic parsing for a voxel-world assistant,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, July 2020. [Online]. Available: <https://www.aclweb.org/anthology/2020.acl-main.427> pp. 4693–4714.
- [107] A. Szlam, J. Gray, K. Srinet, Y. Jernite, A. Joulin, G. Synnaeve, D. Kiela, H. Yu, Z. Chen, S. Goyal, D. Guo, D. Rothmel, C. L. Zitnick, and J. Weston, “Why build an assistant in Minecraft?” *arXiv preprint arXiv:1907.09273*, 2019. [Online]. Available: <http://arxiv.org/abs/1907.09273>
- [108] A. Köhn and A. Koller, “Talking about what is not there: Generating indefinite referring expressions in Minecraft,” in *Proceedings of the 12th International Conference on Natural Language Generation*. Tokyo, Japan: Association for Computational Linguistics, Oct.–Nov. 2019. [Online]. Available: <https://www.aclweb.org/anthology/W19-8601> pp. 1–10.
- [109] A. Köhn, J. Wichlacz, Álvaro Torralba, D. Höller, J. Hoffmann, and A. Koller, “Generating instructions at different levels of abstraction,” in *Proceedings of the 28th International Conference on Computational Linguistics*, Barcelona, Spain, Dec. 2020.
- [110] J. Wichlacz, A. Torralba, and J. Hoffmann, “Construction-planning models in Minecraft,” in *Proc. ICAPS Workshop on Hierarchical Planning*, 2019.
- [111] D. Nau, M. Ghallab, and P. Traverso, *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

- [112] P. Bercher, G. Behnke, D. Höller, and S. Biundo, “An admissible HTN planning heuristic,” in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017. [Online]. Available: <https://doi.org/10.24963/ijcai.2017/68> pp. 480–488.
- [113] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019. [Online]. Available: <https://www.aclweb.org/anthology/N19-1423> pp. 4171–4186.
- [114] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, “BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, July 2020. [Online]. Available: <https://www.aclweb.org/anthology/2020.acl-main.703> pp. 7871–7880.
- [115] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” 2019.
- [116] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” 2020.
- [117] J. Lu, D. Batra, D. Parikh, and S. Lee, “Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks,” in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: <https://proceedings.neurips.cc/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf> pp. 13–23.